The Pragmatic Programmers

Complex Network Analysis in Python

Recognize → Construct → Visualize → Analyze → Interpret



Dmitry Zinoviev

edited by Adaobi Obi Tulton

Early Praise for *Complex Network Analysis in Python*

This book is an excellent read for anyone who wants to learn the fundamentals of complex network analysis with a focus on application. The case studies cover a variety of topics and help readers link concepts to applications, providing readers with a clear, well-structured, hands-on experience that deepens their understanding of the concepts without requiring Python programming experience.

➤ **Kate Li, PhD**
Associate Professor, Sawyer Business School, Suffolk University

As a social scientist interested in network analysis but having limited knowledge of Python, I found the book very useful. The author explains technical problems in a way that is easy to understand for non–computer scientists. It is a great introduction for those interested in network analysis seeking to apply the method in their research.

➤ **Weiqi Zhang**
Associate Professor of Government, Suffolk University

Complex Network Analysis in Python is a thorough introduction to the tools and techniques needed for complex network analysis. Real-world case studies demonstrate how one can easily use powerful Python packages to analyze large networks and derive meaningful analytic insights.

➤ **Mike Lin**
Senior Software Engineer, Fugue, Inc.

Having a deep understanding of complex network analysis is hard; however, this book will help you learn the basics to start mastering the skills you need to analyze complex networks, not only at a conceptual level, but also at a practical level, by putting the theory into action using the Python programming language.

➤ **Jose Arturo Mora**
Head of Information Technology and Innovation, BNN Mexico

Complex networks have diverse applications in various fields, including health care, social networks, and machine learning. I found this book to be an excellent and comprehensive resource guide for researchers, students, and professionals interested in applying complex networks.

➤ **Rajesh Kumar Pandey**
Graduate Student, IIT Hyderabad

Complex Network Analysis in Python

Recognize → Construct → Visualize → Analyze → Interpret

Dmitry Zinoviev

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Adaobi Obi Tulton
Copy Editor: Nicole Abramowitz
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-269-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2018

*To my beautiful and most intelligent wife,
Anna, and to our children: graceful ballerina,
Eugenia, and romantic gamer, Roman.*

Contents

Acknowledgments xi
Preface xiii

1. The Art of Seeing Networks 1
   Know Thy Networks 2
   Enter Complex Network Analysis 5
   Draw Your First Network with Paper and Pencil 6

Part I — Elementary Networks and Tools

2. Surveying the Tools of the Craft 11
   Do Not Weave Your Own Networks 11
   Glance at iGraph 12
   Appreciate the Power of graph-tool 13
   Accept NetworkX 15
   Keep in Mind NetworKit 15
   Compare the Toolkits 16

3. Introducing NetworkX 17
   Construct a Simple Network with NetworkX 17
   Add Attributes 23
   Visualize a Network with Matplotlib 25
   Share and Preserve Networks 29

4. Introducing Gephi 31
   Worth 1,000 Words 31
   Import and Modify a Simple Network with Gephi 32
   Explore the Network 34
   Sketch the Network 36

Prepare a Presentation-Quality Image 38
Combine Gephi and NetworkX 40

5. **Case Study: Constructing a Network of Wikipedia Pages . . . 41**
Get the Data, Build the Network 42
Eliminate Duplicates 45
Truncate the Network 46
Explore the Network 47

Part II — Networks Based on Explicit Relationships

6. **Understanding Social Networks 53**
Understand Egocentric and Sociocentric Networks 53
Recognize Communication Networks 61
Appreciate Synthetic Networks 63
Distinguish Strong and Weak Ties 66

7. **Mastering Advanced Network Construction 69**
Create Networks from Adjacency and Incidence Matrices 69
Work with Edge Lists and Node Dictionaries 76
Generate Synthetic Networks 78
Slice Weighted Networks 79

8. **Measuring Networks 83**
Start with Global Measures 83
Explore Neighborhoods 84
Think in Terms of Paths 88
Choose the Right Centralities 92
Estimate Network Uniformity Through Assortativity 97

9. **Case Study: Panama Papers 101**
Create a Network of Entities and Officers 101
Draw the Network 104
Analyze the Network 105
Build a “Panama” Network with Pandas 108

Part III — Networks Based on Co-Occurrences

10. **Constructing Semantic and Product Networks 115**
Semantic Networks 116
Product Networks 120

11. **Unearthing the Network Structure** 125
Locate Isolates 125
Split Networks into Connected Components 126
Separate Cores, Shells, Coronas, and Crusts 129
Extract Cliques 131
Recognize Clique Communities 134
Outline Modularity-Based Communities 136
Perform Blockmodeling 138
Name Extracted Blocks 139

12. **Case Study: Performing Cultural Domain Analysis** . . . 141
Get the Terms 142
Build the Term Network 146
Slice the Network 147
Extract and Name Term Communities 148
Interpret the Results 150

13. **Case Study: Going from Products to Projects** 153
Read Data 153
Analyze the Networks 155
Name the Components 157

Part IV — Unleashing Similarity

14. **Similarity-Based Networks** 163
Understand Similarity 163
Choose the Right Distance 167

15. **Harnessing Bipartite Networks** 175
Work with Bipartite Networks Directly 176
Project Bipartite Networks 178
Compute Generalized Similarity 181

16. **Case Study: Building a Network of Trauma Types** . . . 185
Embark on Psychological Trauma 185
Read the Data, Build a Bipartite Network 186
Build Four Weighted Networks 188
Plot and Compare the Networks 191

Part V — When Order Makes a Difference

17. Directed Networks **197**

Discover Asymmetric Relationships 197

Explore Directed Networks 199

Apply Topological Sort to Directed Acyclic Graphs 203

Master “toposort” 204

A1. Network Construction, Five Ways **209**

Pure Python 209

iGraph 210

graph-tool 211

NetworkX 212

NetworKit 212

A2. NetworkX 2.0 . . . . **213**

Bibliography **215**

Index **219**

Acknowledgments

This book would not be possible without my editor, Adaobi Obi Tulton. She had the courage to learn the dark inner secrets of complex network analysis and guided me through the minefields of manuscript preparation, from the fuzzy ideas at the onset to this very book in flesh and blood. Thank you, Adaobi.

I am grateful to my reviewers (in alphabetical order): Cody Buntain (University of Maryland), Remy Cazabet (Lyon University), Mark Chu-Carroll (Imagen Technologies), Raphaël Fournier-S'niehotta (CÉDRIC), Michael Lin (Fugue Inc.), Jason Montojo (University of Toronto), Jose Arturo Mora (EY, BNN Mexico), Prasham Ojha (University of Koblenz-Landau), Talha Oz (George Mason University), and Rajesh Kumar Pandey (Gade Autonomous Systems). Your reviews were indispensable. They profoundly affected the book's style, structure, and usability. Thank you, my reviewers.

My wife, Anna; my children, Eugenia and Roman; and my friends and colleagues from Suffolk University provided much-needed emotional support. Writing a book is a quest. It feels good to be well supported. Thank you, my supporters.

Last but not least, the early readers of the beta book provided the errata requests. *Errare humanum est*, but the book is better without errors. Thank you, my early readers.

Thou wilt set forth at once because the journey is far and lasts for many hours; but the hours on the velvet spaces are the hours of the gods, and we may not say what time such an hour may be if reckoned in mortal years.

➤ *Lord Dunsany, Anglo-Irish writer and dramatist*

Preface

In science, technology, and mathematics, a network is a system of interconnected objects. Complex network analysis (CNA) is a discipline of exploring quantitative relationships in the networks with non-trivial, irregular structure. The actual nature of the networks (social, semantic, transportation, communication, economic, and the like) doesn't matter, as long as their organization doesn't reveal any specific patterns. This book was inspired by a decade of CNA practice and research.

Being a professor of mathematics and computer science at Suffolk University in Boston, I have experimented with complex networks of various sizes, purposes, and origins. I developed my first CNA software in an ad hoc manner in the C language—the language venerable yet ill-suited for CNA projects. The price of explicit memory management, cumbersome file input/output, and lack of advanced built-in data structures (such as maps and lists) was simply too high to justify a further commitment to C. At the moment I realized that there were affordable alternatives to C that did not require low-level programming (such as *Pajek [NMB11]* and Mathematica[1]), off I went.

Both systems that I mentioned had significant restrictions. Mathematica was proprietary (and, frankly, quite costly). My inner open source advocate demanded that I cease and desist using it, especially given that earlier versions of Mathematica didn't provide dedicated CNA support and failed to handle big networks. Pajek was proprietary, too, and not programmable. It took a joint effort of my inner open source advocate and inner programmer to push it to the periphery. (I still occasionally use Pajek, and I believe it's a great system for solving non-recurring problems.)

I felt delighted when, in search of open source, free, scalable, reliable, and programmable CNA software, I ran into NetworkX, a Python library still in its infancy. For the next several years, it became my tool of choice when it came to CNA simulation, analysis, or visualization.

1. www.wolfram.com/mathematica

About the Reader

This book is intended for graduate and undergraduate students, complex data analysis (CNA) or social network analysis (SNA) instructors, and CNA/SNA researchers and practitioners. The book assumes that you have some background in computer programming—namely, in Python programming. It expects from you no more than common sense knowledge of complex networks. The intention is to build up your CNA programming skills and at the same time educate you about the elements of CNA itself. If you're an experienced Python programmer, you can devote more attention to the CNA techniques. On the contrary, if you're a network analyst with less than an excellent background in Python programming, your plan should be to move slowly through the dark woods of data frames and list comprehensions and use your CNA intuition to grasp programming concepts.

About the Book

This book covers construction, exploration, analysis, and visualization of complex networks using NetworkX (a Python library), as well as several other Python modules, and Gephi, an interactive environment for network analysts. The book is not an introduction to Python. I assume that you already know the language, at least at the level of a freshman programming course.

The book consists of five parts, each covering specific aspects of complex networks. Each part comes with one or more detailed case studies.

Part I presents an overview of the main Python CNA modules: NetworkX, iGraph, graph-tool, and networkit. It then goes over the construction of very simple networks both programmatically (using NetworkX) and interactively (in Gephi), and it concludes by presenting a network of Wikipedia pages related to complex networks.

In Part II, you'll look into networks based on explicit relationships (such as social networks and communication networks). This part addresses advanced network construction and measurement techniques. The capstone case study —a network of "Panama papers"—illustrates possible money-laundering patterns in Central Asia.

Networks based on spatial and temporal co-occurrences—such as semantic and product networks—are the subject of Part III. The third part also explores macroscopic and mesoscopic complex network structure. It paves the way to network-based cultural domain analysis and a marketing study of Sephora cosmetic products.

If you cannot find any direct or indirect relationships between the items, but still would like to build a network of them, the contents of Part IV come to the rescue. You will learn how to find out if items are similar, and you will convert quantitative similarities into network edges. A network of psychological trauma types is one of the outcomes of the fourth part.

The book concludes with Part V: directed networks with plenty of examples, including a network of qualitative adjectives that you could use in computer games or fiction.

When you finish your journey, you'll be able to identify, sketch (both by hand, in Gephi, and programmatically), transform, analyze, and visualize several types of complex networks. You'll be able to interpret network measures and structure. The book doesn't aim to be a comprehensive CNA reference. Many discipline-specific aspects, such as triadic census, exponential random graph models (ERGMs), and network flows, as well as the whole story of network dynamics (evolution and contagion), have been intentionally left uncharted. The bibliography on page 215 will take you to more destinations of your choice, whether they be economic networks, web scrapping, or classical social network analysis.

About the Software

This book uses Python 3.x and networkx 1.11. All Python examples in this book are known to work for the modules mentioned in the following table. All of these modules are included in the Anaconda distribution, with the exception of community,[2] toposort,[3] wikipedia,[4] and generalized,[5] which must be installed separately. Anaconda is provided by Continuum Analytics and is available for free.[6]

Package	Used version	Package	Used version
python	3.4.5	networkx	1.11
matplotlib	1.5.1	community	0.9
nltk	3.2.2	numpy	1.11.3
pandas	0.19.2	pygraphviz	1.3.1
wikipedia	1.4	scipy	0.18.1
toposort	1.5		

2. pypi.python.org/pypi/python-louvain
3. pypi.python.org/pypi/toposort
4. pypi.python.org/pypi/wikipedia
5. pragprog.com/titles/dzcnapy/source_code
6. www.continuum.io

The easiest way to install the missing modules is by running pip on your operating system shell command line.

```
⇒ pip install toposort
⇒ pip install wikipedia
⇒ pip install python-louvain
⇒ pip install pygraphviz
```

If you want to use module pygraphviz to layout networks, you first need to install Graphviz (including the developers add-on graphviz-dev).[7]

In September 2017, a new version of NetworkX was released, NetworkX 2.0. Appendix 2, *NetworkX 2.0*, on page 213 provides useful information about converting your CNA scripts to the new version.

About the Notation

The following covers the specific notation used in this book.

Program Output

The book uses a left-pointed gray arrow in the left margin of a page to indicate program outputs. In the following scenario, print(1 + 2) is a Python statement, and 3 is the visual output of the statement.

```
print(1 + 2)
```

```
❮ 3
```

"This Chapter Uses X"

This chapter uses X

"This chapter/section uses X" informs you that the material in the chapter or section goes beyond the core Python and NetworkX. If you're unfamiliar with X, you'll probably understand the content but may experience difficulties with comprehending the included code snippets. You're advised to refresh your knowledge of the listed modules.

Directed Edges

NetworkX uses module Matplotlib for network visualization. You would expect directed edges to have an arrow at the head end, and Matplotlib fully supports arrows. However, NetworkX draws thick rectangular stubs instead. This is just something you'll have to get used to. If you need a publication-quality network image with arrows, consider using Gephi.

7. www.graphviz.org/

Online Resources

This book has its own web page[8] where you can find all the code for this book. There you'll also find the community forum, where you can ask questions, post comments, and submit errata.

Two other great community-operated resources for questions and answers are the Stack Overflow forum[9] and NetworkX Google discussion group.[10]

Now, let's get started!

Dmitry Zinoviev
dzinoviev@gmail.com
January 2018

8. pragprog.com/book/dzcnapy
9. stackoverflow.com/questions/tagged/networkx
10. groups.google.com/forum/#!forum/networkx-discuss

When all you have is a hammer, everything looks like a nail.

> *Proverb*

CHAPTER 1

The Art of Seeing Networks

Complex network analysis (CNA) is a rapidly expanding discipline that studies how to recognize, describe, analyze, and visualize complex networks. The Python library `NetworkX` provides a collection of functions for constructing, measuring, and drawing complex networks. We'll see in this book how CNA and `NetworkX` work together to automate mundane and tedious CNA tasks and make it possible to study complex networks of varying sizes and at varying levels of detail.

At this point, you may be wondering what a network is, why some networks are complex, why it is important to recognize, describe, analyze, and visualize them, and why the discipline is expanding right now instead of having expanded, say, a hundred years ago. If you're not, then you're probably a seasoned complex network researcher, and you may want to skip the rest of this chapter and proceed to the CNA and Python technicalities (Chapter 2, *Surveying the Tools of the Craft*, on page 11). Otherwise, stay with us!

Complex networks, like mathematics, physics, and biology, have been in existence for at least as long as we humans have. Biological complex networks, in fact, predate humankind. However, intensive studies of complex networks did not start until the late 1800s to early 1900s, mostly because of the lack of proper mathematical apparatus (graph theory, in the first place) and adequate computational tools. The reason for the explosion of CNA research and applications in the late 1900s–early 2000s is two-fold. On the "supply" side, it is the availability of cheap and powerful computers and the abundance of researchers with advanced training in mathematics, physics, and social sciences. On the "demand" side, it is the ever increasing complexity of social, behavioral, biological, financial, and technological (to name a few) aspects of humanity.

In this chapter, you will see different types and kinds of networks (including complex networks) and learn why networks are important and why it is worth

seeing them around. You will be able to spot complex networks, capture them —so far, without any software—and get some sense about their useful properties (again, with no software necessary). When you see the limitations of the paper-and-pencil method, you will be ready to dive into the computerized proper complex network analysis.

Know Thy Networks

In general, a network is yet another—relational—form of organization and representation of discrete data. (The other one being tabular, with the data organized in rows and columns.) Two important network concepts are entities and the relationships between them. Depending on a researcher's background, entities are known as nodes (the term we'll use in this book), actors, or vertices. Relationships are known as edges (preferred in this book), links, arcs, or connections. We will casually refer to networks as "graphs" (in the graph-theoretical meaning of the word), even though graphs are not the only way to describe networks.

Graphs and Graphs

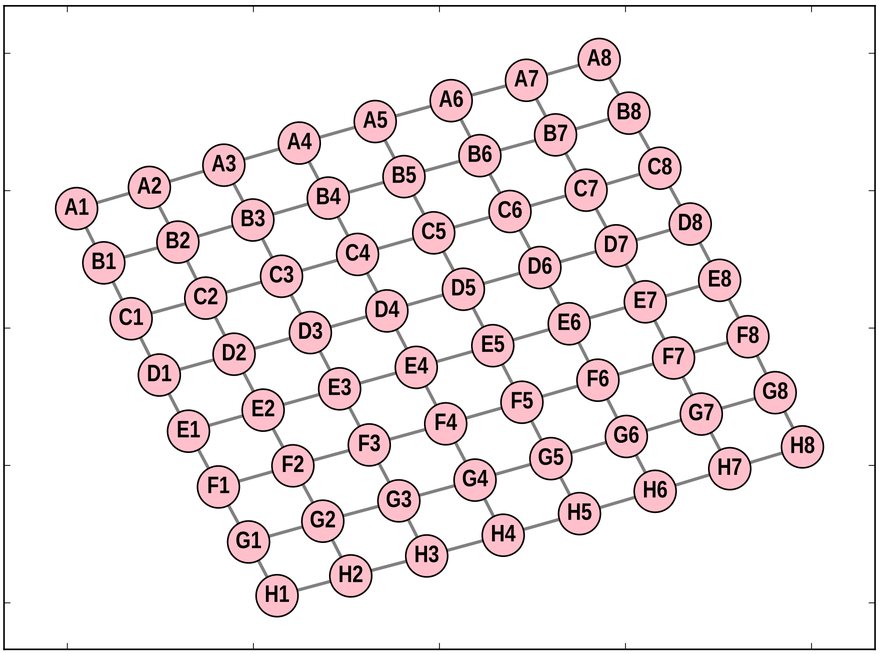
When it comes to mathematics, the word "graph" has at least two different meanings. In algebra and calculus, a graph of a function is a continuous line chart or surface plot. In graph theory, a graph is a set of discrete objects (vertices, depicted diagrammatically as dots), possibly joined by edges (depicted as lines or arcs). We will always use the latter definition unless explicitly stated.

Network nodes and edges are high-level abstractions. For many types of network analysis, their true nature is not essential. (When it is, we decorate nodes and edges by adding properties, also known as attributes.) What matters is the discreteness of the entities and the binarity of the relationships. A discrete entity must be separable from all other entities—otherwise, it is not clear how to represent it as a node. A relationship typically involves two discrete entities; in other words, any two entities either are in a relationship or not. (An entity can be in a relationship with itself. Such a relationship is called *reflexive*.) It is not directly possible to use networks to model relationships that involve more than two entities, but if such modeling is really necessary, then you can use hypergraphs, which are beyond the scope of this book.

Once all of the above conditions are met, you can graphically represent and visualize a node as a point or circle and an edge as a line or arc segment. You can further express node and edge attributes by adding line thickness, color, different shapes and sizes, and the like.

Let's have a look at some really basic—so-called "classic"—networks.

In a checkerboard, each field is an entity (node) with three attributes: "color" ("black" or white"), "column" ("A" through "H"), and "row" (1 through 8). "Being next to" is the relationship between two entities. There is an edge connecting two nodes if the nodes "are next to" each other. As a matter of fact, "being next to" is one of the foundational relationships that leads to spatial networks. You can see a "checkerboard" network, also known as a mesh or grid, in the following figure.

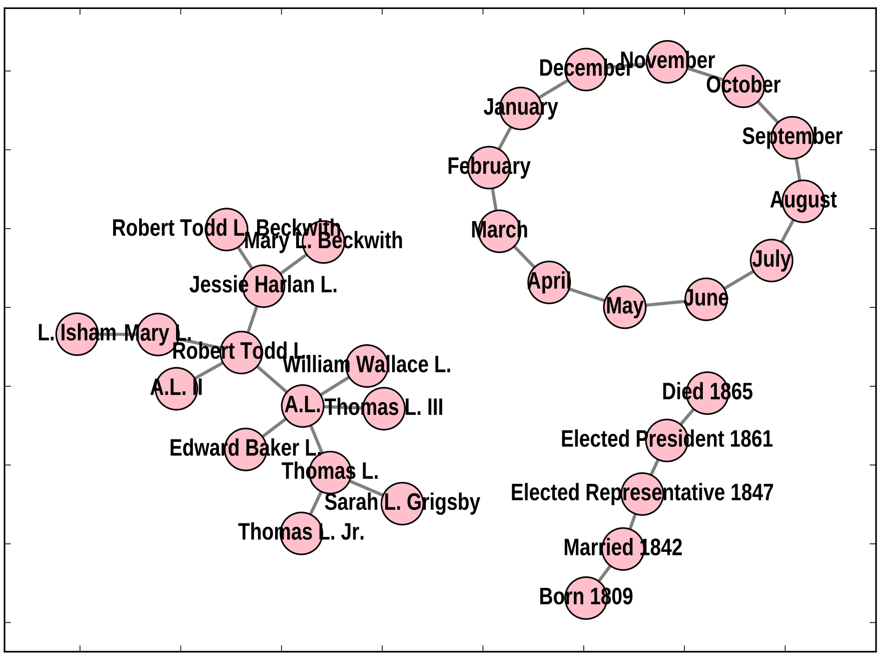


In a timeline of our life, each life event (such as "birth," "high school graduation," "marriage," and eventually "death") is an entity with at least one attribute: "time." "Happening immediately after" is the relationship: an edge connects two events if one event occurs immediately after the other, leading to a network of events. Unlike "being next to," "happening immediately after" is not symmetric: if A happened immediately after B (there is an edge from A to B), then B did not happen after A (there is no reverse edge).

In a family tree, each person in the tree is an entity, and the relationship could be either being "a descendant of" or "an ancestor of" (asymmetric). A family tree network is neither spatial nor strictly temporal: the nodes are not intrinsically arranged in space or time.

In a hierarchical system that consists of parts, sub-parts, and sub-sub-parts (such as this book), a part at any level of the hierarchy is an entity. The relationship between the entities is "a part of": a paragraph is "a part of" a subsection, which is "a part of" a section, which is "a part of" a chapter, which is "a part of" a book.

All the networks listed previously are simple because they have a regular or almost regular structure. A checkerboard is a rectangular grid. A timeline is a linear network. A family tree is a tree, and such is a network of a hierarchical system (a special case of a tree with just one level of branches is called a star). The following figure shows more simple networks: a linear timeline of Abraham Lincoln (A.L.), his family tree, and a ring of months in a year. (A ring is another simple network, which is essentially a linear network, wrapped around.)



Make no mistake: a simple network is simple not because it is small, but because it is regular. For example, any ring node always has two neighbors; any tree node (except for the root) has exactly one antecedent; any inner grid node has exactly four neighbors, two of which are in the same row and the other two in the same column. The complete world timeline has billions of events. The humankind "family tree" has billions of individuals. We still consider these networks simple.

What is a complex network, then?

A complex network has a non-trivial structure. It is not a grid, not a tree, not a ring—but it is not entirely random, either. Complex networks emerge in nature and the man-made world as a result of decentralized processes with no global control. One of the most common mechanisms is the preferential attachment (*Emergence of Scaling in Random Networks [BA99]*), whereby nodes with more edges get even more edges, forming gigantic hubs in the core, surrounded by the poorly connected periphery. Another evolutionary mechanism is transitive closure, which connects two nodes together if they are already connected to a common neighbor, leading to densely interconnected network neighborhoods.

Let's glance at some complex networks. The following table shows the major classes of complex networks and some representatives from each class.

Technological networks	Communication systems; transportation; the Internet; electric grid; water mains
Biological/ecological networks	Food webs; gene/protein interactions; neural system; disease epidemics
Economic networks	Financial transactions; corporate partnerships; international trade; market basket analysis
Social networks	Families and friends; email/SMS exchanges; professional groups
Cultural networks	Language families; semantic networks; literature, art, history, religion networks (emerging fields)

The networks in the table pertain to diverse physical, social, and informational aspects of human life. They consist of various nodes and edges, some material and some purely abstract. However, all of them have common properties and behaviors that can be found in complex networks and only in complex networks, such as community structure, evolution by preferential attachment, and power law degree distribution.

Enter Complex Network Analysis

Complex network analysis (CNA), which is the study of complex networks—their structure, properties, and dynamics—is a relatively new discipline, but with a rich history.

You can think of CNA as a generalization of social network analysis (SNA) to include non-social networks.

Social networks—descriptors of social structures through interactions—have been known as "social groups" since the late 1890s. Their systematic exploration began in the 1930s. In 1934, J.L. Moreno (*Who Shall Survive? [Mor34]*)

developed sociograms—graph drawings of social networks. Eventually, sociograms became the de facto standard of complex network visualization.

John Barnes coined the term "SNA" in 1954 (*Class and Committees in a Norwegian Island Parish [Bar54]*). Around the same time, rapid penetration of mathematical methods into social sciences began, leading to the emergence of SNA as one of the leading paradigms in contemporary sociology.

Social network analysis addresses social networks at three levels: microscopic, mesoscopic, and macroscopic. At the microscopic level, we view a network as an assembly of individual nodes, dyads (pairs of connected nodes; essentially, edges), triads (triples of nodes, connected in a triangular way), and subsets (tightly knit groups of nodes). A mesoscopic view focuses on exponential random graph models (ERGMs), scale-free and small-world networks, and network evolution. Finally, at the macroscopic level, the more general complex network analysis fully absorbs SNA, abstracting from the social origins of social networks and concentrating on the properties of very large real-world graphs, such as degree distribution, assortativity, and hierarchical structure (*Exploring Complex Networks [Str01]*). You will see the definitions and explanations of some of these properties and the Python ways of calculating them later in the book.

But first, let's get your hands dirty (possibly physically dirty) and sketch a real complex network on a sheet of paper.

Draw Your First Network with Paper and Pencil

Just like networks with regular topology, complex networks are not necessarily large. In fact, they are not even "complex" in the colloquial meaning of the word. We can easily spot them without any specialized hardware or software; a pair of inquisitive eyes, a sheet of paper, and a pencil often suffice.

As a proof of concept, let's do an exercise in network construction (just construction, no analysis so far!). We are deeply convinced that complex networks are everywhere; rephrasing the quote, incorrectly attributed to Michelangelo, "all we have to do is to chip away everything that is not a complex network."

All people on Earth, including current and prospective complex network analysts, deserve healthy nutrition. To help them build a balanced diet in an utterly networked way, you will use a list of foods that provide naturally occurring nutrients.[1] The data on the website is somewhat contradictory,

1. The document was originally found at www.sharecare.com/health/nutrition-diet/which-foods-naturally-occurring-nutrients but does not seem to be there anymore; it is cached as nutrients.txt at pragprog.com/book/dzcnapy.

as is often the case with real-world data. For example, in one list item, the authors refer to "shellfish," and in another, to "seafood." It is not clear if freshwater crayfish is meant to be "seafood" or not, but let us not worry about the strict biological taxonomy and make reasonable assumptions, whenever necessary.

Your first step is to identify discrete entities. The dataset has two potential candidates for entities (and, therefore, network nodes): foods (such as fish and eggs) and nutrients (such as vitamins A and C). You could construct a network of foods or a network of nutrients. However, you can shoot two birds with one stone and create a network of both nutrients and foods (a so-called bipartite network—more on them in Chapter 15, *Harnessing Bipartite Networks*, on page 175). The nodes will be of two types, but don't worry about this heterogeneity now.

The relationship between digestive items is described by the verb "provides" or "is provided": certain food X provides nutrients Y1, Y2, and so on, and certain nutrient Y is provided by certain foods X1, X2, and so on.

Now, take a sheet of paper and a pencil and transcribe the list of food and nutrient items into a network, as follows:

1. Choose the first nutrient from the list—say, it is vitamin D. Draw a circle that represents vitamin D and label it "D."

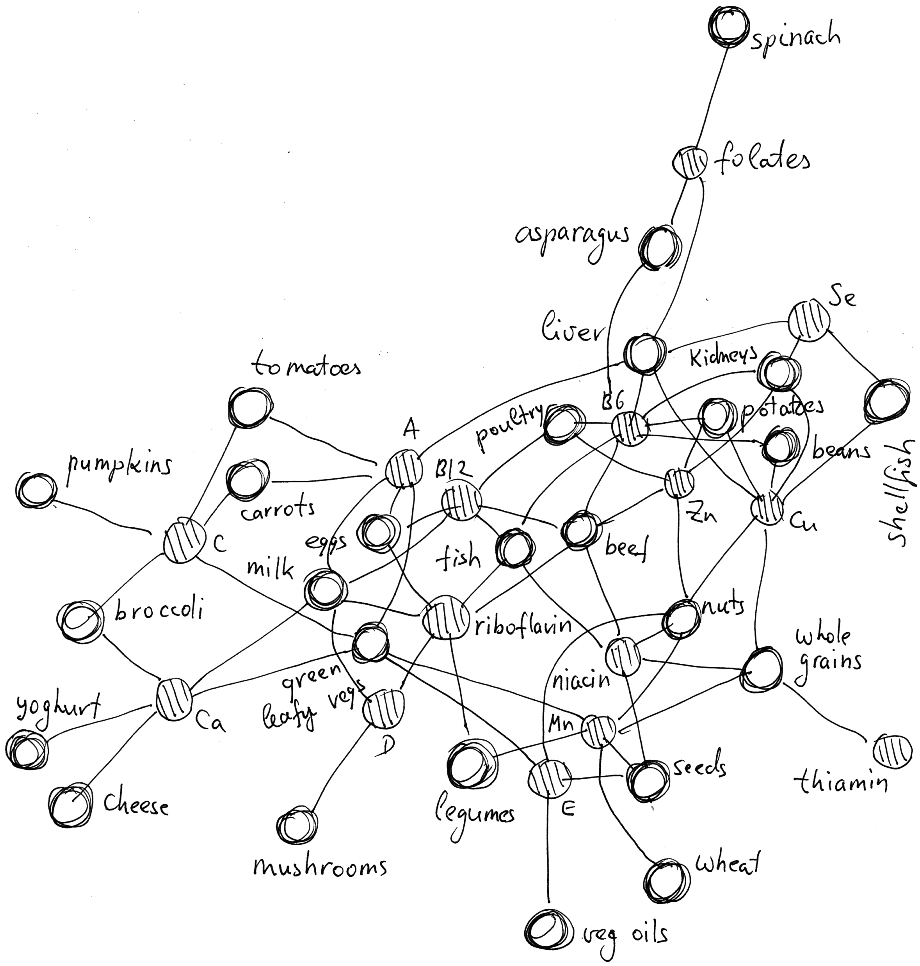
2. Vitamin D is provided by fatty fish; draw a circle that represents fatty fish, label it "fatty fish," and connect to the "D" node.

3. Vitamin D is also provided by mushrooms; draw a circle that represents mushrooms, label it "mushrooms," and connect to the "D" node.

4. Repeat the previous steps for each combination of food types and nutrients. Do not duplicate nodes! If a nutrient is provided by the food type that already has a node, connect the nutrient to the existing node.

The method of starting with a "seed" node and following the edges to discover other nodes is called snowball sampling ("snowballing"). Your network starts as a single snowflake and grows over time until either you are happy with its size or there is no more "snow" to add. Beware: snowballing may overlook small and medium-size network chunks if you choose an improper seed. To mitigate potential problems in networks that consist of several disjointed parts (so-called unconnected graphs), it might be best to select several seeds and follow all edges originating from them.

By the way, congratulations! You just created your first complex network! (Apologies if it was not your first.) Does it look like the following figure?



Is the paper and pencil method tedious? You bet!

Is it error prone? Absolutely!

Is the network drawing ugly? Most likely!

But don't worry. You will see how to automate the network construction process soon (in *Construct a Simple Network with NetworkX, on page 17* and *Import and Modify a Simple Network with Gephi, on page 32*). In this chapter, you learned the simple theory of complex networks and quick-and-dirty paper and pencil network construction tricks. Let's proceed to the overview of Python and non-Python power tools for network construction and analysis.

Part I

Elementary Networks and Tools

Even rudimentary automation of complex network analysis leads to significant performance improvement. The results are even more impressive when you deal with many similar networks that have to be analyzed in a similar way. In this part, you will acquire elementary CNA automation skills.

M. Worsaae of Copenhagen, who has been followed by other antiquaries, has even gone so far as to divide the natural history of civilization into three epochs, according to the character of the tools used in each.

➤ *Samuel Smiles, Scottish author and government reformer*

CHAPTER 2

Surveying the Tools of the Craft

The most common Python tools for manipulating and processing networks are NetworkX, iGraph, graph-tool, and networkit. The modules make it possible to construct complex networks from non-network data, analyze and visualize the networks, and convert the analysis results into non-network data structures (such as dictionaries, lists, and Pandas DataFrames)—in other words, embed CNA into the general-purpose software development workflow. In this chapter, you will look at the four modules and compare their strong and weak points. You will be able to decide if NetworkX, the module that we use in the rest of the book, is right for your problem. (If not, you can still read the coding-agnostic part of the book!) You will be ready to tackle NetworkX and write code to construct "simple" complex networks.

Do Not Weave Your Own Networks

Being a Python programmer, it's often tempting to disregard existing CNA modules (especially since they're not part of the language core) and produce roll-your-own CNA code. For example, you could represent a network as a list of edges (an edge list) or as a dictionary with nodes and edges. You could spend a fortune designing an efficient data structure for the internal network representation. But then the real job begins: implementing dozens of network construction, serialization, deserialization, and analysis algorithms, followed by aesthetically appealing, presentation-quality network visualization.

Even if you're the right person for this task, as a complex network analyst, you want the tools of the craft now, not in weeks or months. You want tools that are bug-free, efficient, and well-documented. You want tools with a broad user base from whom you can seek support and consolation. That is why I encourage you to give up your ambitious plans of building your own CNA suite (if you ever had such plans, of course) and consider one of the existing libraries.

Where to Get Help

Speaking of support and consolation, the primary source of it for a desperate programmer is the Q&A site StackOverflow.[a] The number of individual tags on it attests to the popularity of a library. At the time this book was written, iGraph had 1,940 posts (but only 411 for the Python version); NetworkX was not too far behind with 1,711 posts. The two libraries that support distributed processing via OpenMP—graph-tool and NetworKit—were trailing with 151 and 21 posts. The latter one does not even have a specific tag! Looks like people who work with huge networks know their way around.

a. stackoverflow.com

If you're impatient to see some Python code, Appendix 1, *Network Construction, Five Ways*, on page 209 shows how to construct the same network—the Lincoln graph on page 4—in pure Python and the four packages that you will come to know next.

Glance at iGraph

The library iGraph (properly spelled as iGraph, but imported as igraph) is an open source and free "collection of network analysis tools with the emphasis on efficiency, portability, and ease of use".[1] NetworkX (the tool of choice in the book) and iGraph are structurally similar, yet have their unique features and algorithms.

One of the most notable features of iGraph is its availability as a C language library with the bindings in C, Python, and R. The R application programming interface (API) makes the package a better alternative for the network analysts who have been trained as R programmers. The choice of C as the implementation language also makes the package two orders of magnitude faster than the comparable Python-only packages.

Let's go over the list of iGraph's other niceties. To begin with, the module provides a convenient way to instantiate a network from an edge list—something not critical for most projects, but still nice icing on the cake.

iGraph natively supports node clustering (community detection), which is essential for complex network analysis. While community detection is available for NetworkX through a third-party module (see *Outline Modularity-Based Communities*, on page 136), you may feel more comfortable when it's an integral part of your tool chest.

1. igraph.org

iGraph has a smart built-in search mechanism. A programmer can call methods .select() and .find() with complex queries as parameters to locate nodes and edges, based on their attribute values.

The iGraph drawing subsystem supports a variety of graph layout algorithms that broadly expand presentational opportunities. Once again, NetworkX is capable of similarly complex and perhaps even better charting, but it relies on graphviz (*Harness Graphviz*, on page 28) or similar external programs for node placement. You must install the programs separately, and their versions must match the version of NetworkX. iGraph spares you from the version-matching misery.

Last but not least, iGraph is 10–50 times faster than NetworkX. We already mentioned this fact before, but it is worth mentioning again. The success or failure of your large CNA project may depend on whether you can finish the analysis by the deadline (if at all). If you have a network of more than a hundred thousand nodes, NetworkX may not be your best friend. (Hint: you can still try NetworKit!)

The benefits of iGraph far outweigh its flaws, but it is not flawless. First and foremost, installing the package requires a C compiler and takes considerable time.

Another downside of iGraph is the way it handles nodes (iGraph developers refer to nodes as "vertices") and edges. You can add an edge to a network only if both edge ends have already been added, which is not always desirable. Internally, iGraph stores edges in an indexed list. Any addition or removal of an edge triggers a costly reindexing; for instance, adding a hundred edges one at a time takes roughly a hundred times longer than adding the same edges from a list. What is worse, the removal of edges and nodes changes their indexes. If network elements are removed in a loop, node and edge indexes may be invalidated by prior removals.

Appreciate the Power of graph-tool

graph-tool developers position the module as having "a level of performance that is comparable (both in memory usage and computation time) to that of a pure C/C++ library."[2] Just like in the case of iGraph, the performance boost comes from implementing the whole module in C/C++.

Once successfully installed, graph-tool shines. For starters, it is based on the OpenMP protocol that supports shared memory multiprocessing programming.[3] A graph-tool program is capable of using all CPUs and cores available to your system. Many CNA tasks (such as PageRank and betweenness

2. graph-tool.skewed.de
3. www.openmp.org/

calculation) are easily parallelizable: they can be split into N subtasks, so that each one is executed by a CPU or core, reducing the total running time by the factor of up to N. (Note that even without OpenMP, graph-tool is still the fastest Python CNA library of the four libraries considered.) If you feel your network is too large and CNA jobs take too much time to run, adding another CPU may help.

Here's a list of some of graph-tool's most prominent features:

- Excellent support for drawing. graph-tool supports a variety of layouts and output formats. It can use its built-in layout and visualization engines or rely on the external graphviz package (you will meet it in *Harness Graphviz*, on page 28).
- Extended graph statistics calculation tools that spare you from relying on other statistical modules.
- Built-in community detection and improved blockmodeling (see *Perform Blockmodeling*, on page 138) algorithms. Combined with the OpenMP parallelization, this feature makes graph-tool a really serious community detection engine.
- Graph filtering and graph views. Slicing (*Slice Weighted Networks*, on page 79) is a common task in complex network construction, whereby a decision to keep or discard an edge is made based on its weight. graph-tool allows you to imitate a sliced network graph without really removing the unworthy edges, but by temporarily hiding ("filtering") them, based on their attributes and other properties. The new virtual graphs can be saved as "views" and later analyzed and visualized as if they were genuine networks of their own.

graph-tool's superb performance comes at the cost of increased time and memory required during installation and compilation. My experience shows that installing graph-tool is not even always feasible. If you happen to run Linux and you have not updated it for a while (because "if it ain't broke, don't fix it"), then chances are you will have to compile the module from the C/C++ source code. Some Python programmers hate challenges like that.

graph-tool's other major deficiency is in how it handles nodes (graph-tool calls nodes "vertices."). Unlike other network analysis libraries, graph-tool nodes do not have names. Instead, they have contiguous indexes. (Presumably because graph-tool makes little effort to disguise its C/C++ heritage!) Adding a node to a graph does not affect existing indexes. However, removing a node (unless

it is the node with the largest index) invalidates some or all existing indexes and may cause mysterious errors, especially in loops.

Another inconvenience of graph-tool is related to the separation of nodes and edges and their attributes, including node names/labels. The attributes are stored in dictionary-style data structures called property maps. The node name is just one of the attributes. The programmer is responsible for keeping the node list and the attribute lists in a consistent state. This second-class treatment of node labels makes graph-tool more suitable for general graph analysis, rather than for the analysis of real world–inspired complex networks.

To summarize: graph-tool is a terrific module, though not without issues. Someone should write a book about it, too.

Accept NetworkX

NetworkX *is* indeed the tool of the craft, at least for this book, and I would like to justify my choice.

The main winning points of NetworkX are fourfold:

- NetworkX is painless to install. Since it is written in pure Python, it requires no compilation.
- NetworkX has excellent online documentation, far superior to that of iGraph and graph-tool. Besides, it has an active community of supporters on StackOverflow.[4]
- NetworkX's structure (functions, algorithms, and attributes) are in good agreement with CNA tasks.
- NetworkX's performance is acceptable up to about 100,000 nodes.

While it is true that NetworkX lacks some essential features (such as community detection and advanced visualization layouts), you can easily add these features by installing Python-only third-party modules. We will use some of these modules in this book.

Keep in Mind NetworKit

NetworKit is another great library that supports parallelized network processing. NetworkX and NetworKit are compatible at the graph level. If you are in a rush, construct a network in NetworkX and convert it to NetworKit for further analysis.

4. stackoverflow.com/questions/tagged/networkx

Compare the Toolkits

The following table contains a side-by-side comparison of the toolkits mentioned in the previous sections. The relative slowdown value shows how much *slower* the tool is compared to the fastest tool in the collection (which, incidentally, is graph-tool).

	graph-tool	iGraph	NetworkX	NetworKit
Implementation language	C/C++	C/C++	Python	C/C++
Language bindings	Python	C, Python, R	Python	C++, Python
Installation effort	Hard	Medium	Easy	Medium
OpenMP support	Yes	No	No	Yes
Relative slowdown[5]	1	1–4	40–135	N/A
Built-in community detection	Yes	Yes	No	Yes
Built-in advanced layouts	Yes	Yes	No	Yes

In this chapter, we compared four of the most popular CNA tools written in Python and available for free. You've got to admit that NetworkX does not necessarily look like the best tool. However, it is the most easily installable, the most robust, and the most well documented. It is still a noble and venerable toolset, and we will stick with it. Turn to the next chapter—and you will find out how to create your first NetworkX-based complex network.

5. graph-tool.skewed.de/performance

The result is a most extraordinary looking creature, a network of worms with numerous heads, each branch being eventually provided with one of its own.

➤ *B. Lindsay, American biologist and writer*

CHAPTER 3

Introducing NetworkX

Any network starts with one node, and we can add more nodes and edges to it, as needed. The attributes of those nodes and edges describe their properties. The node, edge, and attribute data come from other data structures or files.

In this chapter, you will learn NetworkX functions for starting a new network, populating it with nodes and edges, and decorating them with attributes. You will also learn how to create a "quick and dirty" visualization of the constructed network (we will give you more powerful visualization tools later in Chapter 4, *Introducing Gephi*, on page 31).

In many cases, complex network analysis is an iterative process, whereby the network grows, shrinks, or undergoes other transformations over time. You will learn how to preserve a complex network as a disk file in a variety of popular formats (some of which you can later import into Gephi, an interactive network analysis tool) and how to read data from appropriate files into the NetworkX representation.

We will use the following terminology throughout the book to refer to the relationships between nodes and edges:

- A node is incident to an edge if it is the start or end of the edge. The edge, respectively, is incident to its end nodes.
- Two nodes are adjacent if they are incident to the same edge.
- Two edges are adjacent if they are incident to the same node.

Construct a Simple Network with NetworkX

A NetworkX project begins with importing module networkx (usually under the name nx).

```
import networkx as nx
```

Create a Graph

A NetworkX network is a collection of edges and labeled nodes. The library allows you to use any hashable Python data as a node label (different labels within the same graph may belong to different data types). To create a new network graph, you must choose an appropriate graph type and call the respective constructor; pass either no parameters (for an empty graph) or a list of edges as node pairs (lists or tuples). NetworkX supports four graph types:

- *Undirected graphs* consist only of undirected edges—edges that can be traversed in either direction so that an edge from A to B is the same as an edge from B to A. Mathematically, undirected graphs represent symmetric relationships: if A is in a relationship with B, then B is also in a relationship with A. For example, sistership and companionship are symmetric relationships, but "being in love with" is not (at least, not always). Create an empty undirected graph with the constructor `nx.Graph()`:

  ```
  G = nx.Graph()
  ```

  Undirected graphs can have self-loops—edges that start and end at the same node. Mathematically, self-loops represent a reflexive relationship: A is in a relationship with itself. If an undirected graph does not have self-loops, it is called a simple graph. A graph that is not simple is called a pseudograph.

- *Directed graphs*, also known as digraphs, have at least one directed edge. "Being the father of" is a symmetric relationship and would be represented by a directed edge. You would use a directed graph for a family network that shows fathership and mothership. Create an empty directed graph with the constructor `nx.DiGraph()`:

  ```
  G = nx.DiGraph()
  ```

  Many NetworkX algorithms refuse to calculate with digraphs. You can convert a digraph into an undirected graph. All directed edges become undirected, and all pairs of two reciprocal edges become single edges. However, remember that the original digraph and the derived undirected graph are different.

  ```
  F = nx.Graph(G) # F is undirected
  ```

- *Multigraphs* are like undirected graphs, but they can have parallel edges —multiple edges between the same nodes. Parallel edges may represent different types of relationships between the nodes. For example, Alice may

be a classmate of Bob, but she also may be his friend. Create an empty multigraph with the constructor nx.MultiGraph():

```
G = nx.MultiGraph()
```

- Finally, *directed multigraphs* are what they say they are: directed graphs with parallel edges. Create an empty directed multigraph with the constructor nx.MultiDiGraph():

```
G = nx.MultiDiGraph()
```

Chapter 17, *Directed Networks*, on page 197 in this book is dedicated to directed networks. Until we get there, unless said otherwise, let's assume that all our networks are undirected and don't have parallel edges, but possibly have self-loops.

Add and Remove Nodes and Edges

NetworkX provides several mechanisms for adding nodes and edges to an existing graph: one by one, from a list or another graph. Likewise, you can remove nodes or edges one by one or by using a list. Node and edge manipulations are subject to the following rules:

- Adding an edge to a graph also ensures that its ends are added if they did not exist before.
- Adding a duplicate node or edge is silently ignored unless the graph is a multigraph; in the latter case, an additional parallel edge is created.
- Removing an edge does not remove its end nodes.
- Removing a node removes all incident edges.
- Removing a single non-existent node or edge raises a NetworkXError exception, but if the node or edge is a part of a list, then an error is silently ignored.

Let's use the data collected in *Draw Your First Network with Paper and Pencil*, on page 6 to build the same network of foods and nutrients programmatically, using all node addition techniques mentioned previously:

```
G = nx.Graph([("A", "eggs"),])
G.add_node("spinach") # Add a single node
G.add_node("Hg") # Add a single node by mistake
G.add_nodes_from(["folates", "asparagus", "liver"]) # Add a list of nodes
G.add_edge("spinach", "folates") # Add one edge, both ends exist
G.add_edge("spinach", "heating oil") # Add one edge by mistake
G.add_edge("liver", "Se") # Add one edge, one end does not exist
G.add_edges_from([("folates", "liver"), ("folates", "asparagus")])
```

We intentionally added several inedible nodes—just to illustrate how one can remove unwanted fragments:

```
G.remove_node("Hg")
G.remove_nodes_from(["Hg",]) # Safe to remove a missing node using a list
G.remove_edge("spinach", "heating oil")
G.remove_edges_from([("spinach", "heating oil"), ]) # See above
G.remove_node("heating oil") # Not removed yet
```

You can use the method G.clear() to delete all graph nodes and edges at once but keep the graph shell.

Look at Edge and Node Lists

NetworkX provides several options for exploring the node and edge lists. Graph object attributes (not to be confused with network attributes in *Add Attributes*, on page 23) G.node and G.edge store all nodes and edges, respectively, in the form of dictionaries. Node labels are the keys of G.node. Node attributes, in the form of nested dictionaries, are the values. Since we did not assign any attributes to the nodes yet, the dictionaries are empty.

```
print(G.node)
```

```
❮ {'Se': {}, 'eggs': {}, 'asparagus': {}, 'A': {}, 'liver': {}, 'spinach': {},
  'folates': {}}
```

Start node labels are the keys of G.edge, too. Each dictionary value corresponds to one edge (also in the form of a dictionary), where the keys are end node labels, and the values are edge attribute dictionaries.

```
print(G.edge)
```

```
❮ {'Se': {'liver': {}}, 'eggs': {'A': {}}, 'asparagus': {'folates': {}},
  'A': {'eggs': {}}, 'liver': {'Se': {}, 'folates': {}},
  'spinach': {'folates': {}},
  'folates': {'asparagus': {}, 'liver': {}, 'spinach': {}}}
```

The second option is to call methods G.nodes() and G.edges(). (Mind the s at the end!) If called without any parameters, the methods return node and edge lists.

```
print(G.nodes())
```

```
❮ ['Se', 'eggs', 'asparagus', 'A', 'liver', 'spinach', 'folates']
```

```
print(G.edges())
```

```
❮ [('Se', 'liver'), ('eggs', 'A'), ('asparagus', 'folates'),
  ('liver', 'folates'), ('spinach', 'folates')]
```

Note NetworkX created edge attribute accessors G.edge['Se']['liver'] and G.edge['liver']['Se'], but the edge ('Se', 'liver') does not have the reverse counterpart ('liver', 'Se').

If called with the optional parameter data=True, the methods return the lists with the additional attribute dictionaries.

```
print(G.nodes(data=True))
```

```
❮ [('Se', {}), ('eggs', {}), ('asparagus', {}), ('A', {}), ('liver', {}),
   ('spinach', {}), ('folates', {})]
```

```
print(G.edges(data=True))
```

```
❮ [('Se', 'liver', {}), ('eggs', 'A', {}), ('asparagus', 'folates', {}),
   ('liver', 'folates', {}), ('spinach', 'folates', {})]
```

You can measure the length of the returned lists or dictionaries to find out the number of nodes and edges. Additionally, function len(G) returns the number of nodes in G.

Read a Network from a CSV File

The toy code fragment on page 19 has at least three problems. First, it is a toy. Second, it is incomplete. Third, it is not flexible: any change in the original network requires that you rewrite the code.

Ideally, you would record a network in a file (using some popular data format, such as comma-separated values, or CSV). You would then write a program that reads the network data from the file and constructs a Graph object. NetworkX has an excellent collection of file readers and writers, but let's pretend it does not and implement a CSV edge list reader. Our nutrients and foods are in the file nutrients.csv.[1] The first ten and the last five lines of the file are shown below:

```
A,carrots
A,eggs
A,"fatty fish"
A,"green leafy vegs"
A,liver
A,milk
A,tomatoes
B12,milk
B6,asparagus
B6,beans
«more pairs»
shellfish,Se
thiamin,"whole grains"
tomatoes,tomatoes
"veg oils",E
yogurt,Ca
```

1. pragprog.com/titles/dzcnapy/source_code

Now, watch the magic of Python. It takes only three lines of code to open the edge list file, create a CSV reader for the file, and “suck” the list of pairs into the Graph constructor.

nutrients.py

```
import networkx as nx
import matplotlib.pyplot as plt
import dzcnapy_plotlib as dzcnapy
import csv

with open("nutrients.csv") as infile:
    csv_reader = csv.reader(infile)
    G = nx.Graph(csv_reader)
print(G.nodes())
```

```
❮ ['B6', 'wheat', 'nuts', 'beef', 'cheese', 'milk', 'B12', 'E', 'thiamin',
  'liver', 'legumes', 'broccoli', 'C', 'folates', 'yogurt', 'tomatoes',
  'veg oils', 'riboflavin', 'beans', 'mushrooms', 'D', 'spinach', 'shellfish',
  'niacin', 'A', 'fatty fish', 'Se', 'Mn', 'green leafy vegs', 'poultry',
  'pumpkins', 'Cu', 'whole grains', 'Zn', 'eggs', 'carrots', 'asparagus',
  'potatoes', 'Ca', 'kidneys', 'seeds']
```

The provided edge list in the file nutrients.csv has an intentional inconsistency: an edge that connects the node “tomatoes” with itself, a self-loop. You can remove the self-loops by first identifying them with G.selfloop_edges() and then passing the loop edges to G.remove_edges_from():

nutrients.py

```
loops = G.selfloop_edges()
G.remove_edges_from(loops)
print(loops)
```

```
❮ [('tomatoes', 'tomatoes')]
```

```
loops = G.selfloop_edges()
print(loops) # No more loops
```

```
❮ []
```

Relabel Nodes

The network looks magnificent, but there is one more thing we can do to make it better: capitalize all node names. NetworkX provides method nx.relabel_nodes() that takes a graph and a dictionary of old and new labels and either creates a relabeled copy of the graph (copy=True, default) or modifies the graph in place (use the latter option if the graph is large and you don’t plan to keep the original graph). Each dictionary key must be an existing node label, but some labels may be missing. The respective nodes will not be relabeled.

We will use dictionary comprehension to walk through all network nodes and convert those labeled with strings to the title case (capitalize the first letter of each word).

nutrients.py

```
mapping = {node: node.title() for node in G if isinstance(node, str)}
nx.relabel_nodes(G, mapping, copy=False)
print(G.nodes())
```

```
❮ ['B6', 'Yogurt', 'Legumes', 'Tomatoes', 'Potatoes', 'Wheat', 'Eggs',
  'Veg Oils', 'D', 'Pumpkins', 'Poultry', 'Kidneys', 'Liver', 'Broccoli',
  'Zn', 'Carrots', 'Whole Grains', 'Folates', 'Niacin', 'Nuts', 'Seeds',
  'Mn', 'C', 'Mushrooms', 'Shellfish', 'B12', 'Cheese', 'Fatty Fish', 'E',
  'Thiamin', 'Riboflavin', 'A', 'Green Leafy Vegs', 'Se', 'Beef',
  'Asparagus', 'Milk', 'Cu', 'Spinach', 'Beans', 'Ca']
```

Note that `G` in the previous code fragment acts as a node iterator. In fact, `G` has some other `dict()` features. For example, you can use selection operator [] to access the edges incident to the node, and their attributes:

```
print(G["Zn"])
```

```
❮ {'Liver': {}, 'Nuts': {}, 'Beef': {}, 'Beans': {}, 'Poultry': {},
  'Potatoes': {}, 'Kidneys': {}}
```

But wait, the network does not have any attributes yet. You may want to add them now.

Add Attributes

A node or edge attribute describes its non-structural properties. For example, edge attributes may represent weight, strength, or throughput. Node attributes may represent edge, color, size, or gender. NetworkX provides mechanisms for setting, changing, and comparing attributes.

An attribute is implemented as a dictionary associated with the node or edge. The dictionary keys are attribute names. As such, they must be immutable: `int()`, `float()`, `bool()`, `str()`, and so on. There are no limitations on the values. You can create a node whose attribute is the node itself, except that this exercise is utterly pointless.

NetworkX offers three options for setting node and edge attributes.

- Define attributes at the time of adding nodes or edges:

  ```
  G.add_node("Honey", edible=True)
  G.add_nodes_from([("Steel", {"edible" : False}), ])
  G.add_edge("Honey", "Steel", weight=0.0)
  G.add_edges_from([("Honey", "Zn"),], related=False)
  ```

NetworkX and other CNA libraries consistently use edge attribute "weight" to denote edge strength (as in *Distinguish Strong and Weak Ties*, on page 66). A graph whose edges have this attribute is called a weighted graph. There is a method G.add_weighted_edges_from() for adding weighted edges.

```
G.add_weighted_edges_from([("Honey", "Zn", 0.01),
                           ("Honey", "Sugar", 0.99)])
```

Note that when you add several edges with G.add_edges_from(), you can specify only one set of attributes for all of them. If that's not what you want, use the next option.

- Define or change an attribute of existing nodes and edges by calling nx.set_node_attributes() or nx.set_edge_attributes():

```
nx.set_node_attributes(G, att_name, node_dict)
nx.set_edge_attributes(G, att_name, edge_dict)
```

Here, att_name is the name of the affected attribute, node_dict/edge_dict is a dictionary whose keys are existing node labels or edge pairs, and values are attribute values for the respective nodes/edges. If the attribute doesn't exist yet, it's created; otherwise, the value of the existing attribute is changed. If a key isn't a node label or edge pair, the methods raise a KeyError exception.

- Define or change an attribute of individual existing nodes and edges directly through the dictionary interfaces G.node (indexed by node labels) and G.edge (double indexed by start and end node labels):

```
G.node["Zn"]["nutrient"] = True # Zinc is a nutrient
G.edge["Zn"]["Beef"]["weight"] = 0.95 # Zinc and beef are well connected
```

The dictionary interface allows you to remove unwanted attributes:

```
del G.node["Zn"]["nutrient"]
del G.edge["Zn"]["Beef"]["weight"]
```

Regardless of how you create an attribute, it can be modified using any of the three options listed previously.

In our little food and nutrition exercise, we have nodes of two types: foods and nutrients. Labeling them for future analysis would be helpful. Let's create a boolean attribute "nutrient" that is true for nutrients and false for foods. The information about node type was not in the original dataset.

nutrients.py

```
nutrients = set(("B12", "Zn", "D", "B6", "A", "Se", "Cu", "Folates",
                 "Ca", "Mn", "Thiamin", "Riboflavin", "C", "E", "Niacin"))
nutrient_dict = {node: (node in nutrients) for node in G}
nx.set_node_attributes(G, "nutrient", nutrient_dict)
```

Ready, Set(), Go

Python programmers often use collections of items solely for lookup. They are interested in whether an item is in the collection or not, not in where in the collection it is. Python lists have notoriously slow (linear) lookup performance. Whenever possible, convert them into sets that have constant lookup time.

All nodes have been successfully labeled:

```
print(G.nodes(data=True))
```

```
[('Seeds', {'nutrient': False}), ('B12', {'nutrient': True}),
 ('B6', {'nutrient': True}), ('Se', {'nutrient': True}),
 ('Cu', {'nutrient': True}), ('Asparagus', {'nutrient': False}),
 ('Broccoli', {'nutrient': False}), ('Poultry', {'nutrient': False}),
 ('Eggs', {'nutrient': False}), ('D', {'nutrient': True}),
 ('Ca', {'nutrient': True}), ('Whole Grains', {'nutrient': False}),
 ('Beef', {'nutrient': False}), ('Thiamin', {'nutrient': True}),
 ('Shellfish', {'nutrient': False}), ('Kidneys', {'nutrient': False}),
 ('Riboflavin', {'nutrient': True}), ('Spinach', {'nutrient': False}),
 ('Cheese', {'nutrient': False}), ('Beans', {'nutrient': False}),
 ('C', {'nutrient': True}), ('Veg Oils', {'nutrient': False}),
 ('Tomatoes', {'nutrient': False}), ('E', {'nutrient': True}),
 ('Mushrooms', {'nutrient': False}), ('Liver', {'nutrient': False}),
 ('Zn', {'nutrient': True}), ('Niacin', {'nutrient': True}),
 ('A', {'nutrient': True}), ('Folates', {'nutrient': True}),
 ('Legumes', {'nutrient': False}), ('Yogurt', {'nutrient': False}),
 ('Nuts', {'nutrient': False}), ('Mn', {'nutrient': True}),
 ('Milk', {'nutrient': False}), ('Wheat', {'nutrient': False}),
 ('Green Leafy Vegs', {'nutrient': False}),
 ('Pumpkins', {'nutrient': False}), ('Carrots', {'nutrient': False}),
 ('Potatoes', {'nutrient': False}), ('Fatty Fish', {'nutrient': False})]
```

We will get back to the node attributes in *Estimate Network Uniformity Through Assortativity*, on page 97. But how does the network appear?

Visualize a Network with Matplotlib

This section uses Matplotlib.

NetworkX is not aesthetically the best library for network visualization. In fact, it does not even do visualization on its own but uses services rendered by Matplotlib, a multipurpose graphics library. Luckily, the team of NetworkX and Matplotlib is fast and easy to understand, and the interaction with Matplotlib is well hidden from the network analyst; you do not need to learn yet another library—but still, you need to import it.

```
import matplotlib.pyplot as plt
```

Note that you typically do not need the whole library, but just the submodule matplotlib.pyplot.

The process of network visualization consists of two phases: layout and rendering. At the layout phase, the software selects geometric positions of each node according to a layout algorithm.

NetworkX supports a variety of layout algorithms. You can choose one of them, based on your aesthetic preferences and your network's aesthetic propensity. For each algorithm, NetworkX has a proper layout function that takes the graph to plot and returns a dictionary of node positions (to be used at the rendering phase), and an all-in-one function that does both layout and rendering. The following table shows the most useful layout and drawing functions. The figure on page 28 shows some actual layouts.

Layout	Arrange node...	Layout function	All-in-one function
Random	Randomly (this layout requires NumPy)	pos=nx.random_layout()	nx.draw_random()
Circular	On a circle	pos=nx.circular_layout()	nx.draw_circular()
Shell	On concentric circles, as defined by nlist	pos=nx.shell_layout(G,nlist=None)	nx.draw_shell()
Spectral	Based on their eigenvector centrality values (see *Eigenvector Centrality*, on page 94)	pos=nx.spectral_layout()	nx.draw_spectral()
Force-directed	As if they were physical balls that repel one another, connected with springs	pos=nx.fruchterman_reingold_layout()	
Same as above	Same as above		nx.draw_networkx()
Same as above	Same as above	pos=nx.spring_layout()	nx.draw_spring()

At the rendering phase, NetworkX draws the nodes, labels, and edges at the prescribed positions, using the default or specified shapes, fonts, and colors. You can see the graphical output on the screen, save it into a file (supported formats include PNG, PDF, PostScript, EPS, and SVG), or both. In the latter case, you must first save the image and only then display it.

The following code fragment prepares a color sequence (pink vs. yellow, depending on the node type) for the nodes.

nutrients.py
```
# Prepare for drawing
colors = ["yellow" if n[1]["nutrient"] else "pink" for n in
          G.nodes(data=True)]
dzcnapy.medium_attrs["node_color"] = colors
```

For each of the four layouts (specified by the subplot axes, layout method, and human-readable title), we calculate the node positions and call nx.draw_networkx(), the generic drawing method.

nutrients.py
```
# Draw four layouts in four subplots
_, plot = plt.subplots(2, 2)

subplots = plot.reshape(1, 4)[0]
layouts = (nx.random_layout, nx.circular_layout, nx.spring_layout,
           nx.spectral_layout)
titles = ("Random", "Circular", "Force-Directed", "Spectral")
for plot, layout, title in zip(subplots, layouts, titles):
    pos = layout(G)
    nx.draw_networkx(G, pos=pos, ax=plot, with_labels=False,
                     **dzcnapy.medium_attrs)
    plot.set_title(title)
➤   dzcnapy.set_extent(pos, plot)
```

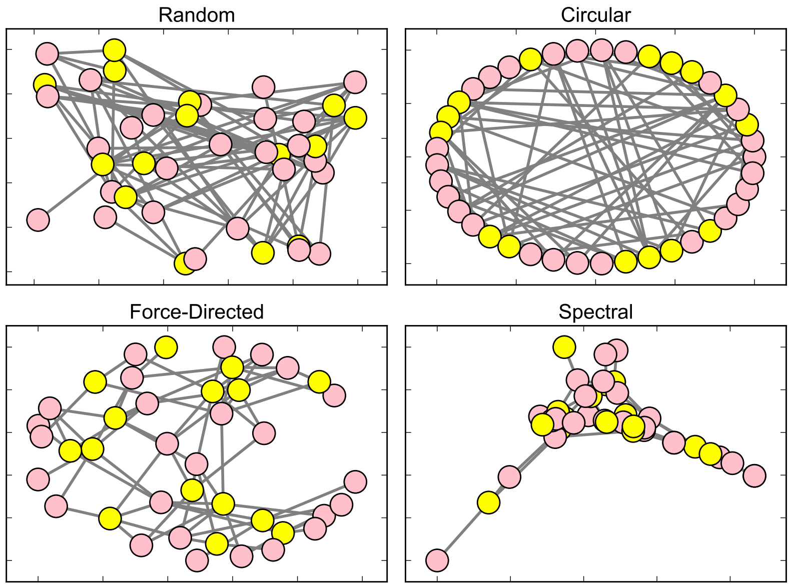
NetworkX doesn't take very good care of scaling network charts. You can do better by manually calculating the extent of each layout and reserving enough space. The highlighted code calls function dzcnapy_plotlib.set_extent(pos, plot) from the module dzcnapy_plotlib.[2] The function fits a network with the nodes at the positions pos into the drawable plot.

Finally, tell Matplotlib to pack the subplots as tight as possible and save and display the images by calling the auxiliary function dzcnapy_plotlib.plot().

nutrients.py
```
dzcnapy.plot("nutrients")
```

The figure on page 28 shows four different drawings of the same network of foods and nutrients. Remember that most of the layout procedures are probabilistic. If you run the code yourself, you will likely get a somewhat different layout.

2. Available from pragprog.com/book/dzcnapy



For most complex networks, the spring layout (the default layout for nx.draw_networkx()) produces the most pleasing output. Note that Matplotlib does not accurately place node labels (to the extent that I preferred not to show them in the figure at all). Some of them badly overlap. If you think this is a problem (and yes, it is!), switch to Gephi—it is described in Chapter 4, *Introducing Gephi,* on page 31. But try graphviz first!

Harness Graphviz

graphviz is an open source graph visualization tool written in C, with bindings available in C, Tcl/Tk, guile, Java, Perl, PHP, Ruby, and, most importantly, Python. Among other things, it provides yet another layout engine, which is typically better than any of the engines mentioned previously.

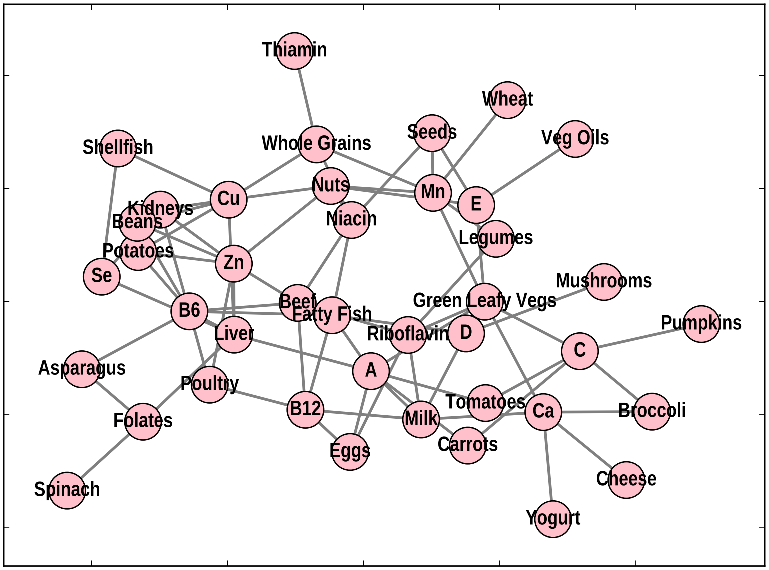
Using graphviz in your code is trivial. The only two lines affected by the switch from, say, nx.draw_networkx() to graphviz are highlighted in the following code fragment. Due to the better overall layout quality, the node labels have better chances of not overlapping and should not be disabled.

nutrients.py

```
➤ from networkx.drawing.nx_agraph import graphviz_layout

  _, plot = plt.subplots()
➤ pos = graphviz_layout(G)
  nx.draw_networkx(G, pos, **dzcnapy.attrs)
  dzcnapy.set_extent(pos, plot)
  dzcnapy.plot("nutrients-graphviz")
```

The following figure shows the output of graphviz. Compare it to the figure on page 8 or to any graph in the previous figure. The difference is impressive.



As you are getting ready to save your network into a file, here's some food for thought: why do all pink nodes have yellow neighbors and the other way around?

Share and Preserve Networks

At this point, you must be very proud of the job well done. The network of foods and nutrients has been extracted, constructed, and visualized. It's time to save it into a file, and there are several compelling reasons for doing so:

1. You never analyzed the network, because you don't know how. When you read through Chapter 8, *Measuring Networks,* on page 83 and Chapter 11, *Unearthing the Network Structure,* on page 125, you'll be able to calculate various network measures and extract network structure. You'll need the network when you get there, but you won't have it unless you save it now.
2. You may want to get a better network visualization of the network with Gephi. The only way for NetworkX to pass the network to Gephi is via a file.
3. Sharing the network with the fellow researchers is priceless, but they want a file with the live network edges and nodes, not a still image.

NetworkX supports many popular file formats suitable for interchanging network data with other software.

Export and Import Networks

Any NetworkX network can be exported to or imported from files (*serialized* and *de-serialized*) in the formats shown in the following table. All nx.read__() functions take the name of an existing file or an open file handle and return a Graph object. All nx.write__() functions take a Graph object and the name of an existing file or an open file handle. Files with names ending in .gz or .bz2 are automatically compressed or uncompressed. Some functions require that the files be opened in the binary mode.

Format	Attributes	Reader	Writer	Supported by Gephi?
Adjacency list	Not stored	nx.read_adjlist()	nx.write_adjlist()	Yes
Edge list	Not stored	nx.read_edgelist()	nx.write_edgelist()	Yes
Graph exchange XML format	Stored	nx.read_gexf()	nx.write_gexf()	Yes
Graph modeling language	Stored	nx.read_gml()	nx.write_gml()	W/o attributes
GraphML	Stored	nx.read_graphml()	nx.write_graphml()	Yes
Pajek NET	Not stored	nx.read_pajek()	nx.write_pajek()	Yes
Pickle	Stored	nx.read_gpickle()	nx.write_gpickle()	No
YAML	Stored	nx.read_yaml()	nx.write_yaml()	No

As an example, let's export the G network as a GraphML file. In my experience, GraphML is the best interchange format between NetworkX and Gephi.

```
nx.write_graphml(G, "nutrients.graphml")
```

Or:

```
with open("nutrients.graphml", "wb") as ofile:
    nx.write_graphml(G, ofile)
```

You learned about the foundations of NetworkX—a powerful Python library for network analysis and visualization. You know how to construct a simple graph incrementally, add node attributes, do some simple visualization, and save the networks to and restore from files. Of these tasks, it is visualization that NetworkX does not handle well.

You've heard so much about Gephi so far that you should not be surprised that it is hiding around the corner [of this page]. Just for one chapter, let's set NetworkX aside and look into this interactive CNA tool. Sometimes, Gephi is the quickest way to analyze a not-so-large network once or twice.

Visual signaling is and always will be a most valuable means of transmitting information in peace and war, and it is not to be imagined that it will ever be supplanted in its particular function by the introduction of other methods.

➤ *Signal Corps United States Army*

CHAPTER 4

Introducing Gephi

When you explore an unfamiliar complex network for the first time, it often helps to perform a quick visual check of its structure before engaging in expensive code writing. Sometimes, you can semi-automate even the network construction itself (we are talking about small networks, with fewer than a couple dozen nodes).

You can perform many one-time construction, analysis, and conversion tasks with Gephi. Gephi is a free, Java-based, interactive CNA environment that runs on all mainstream operating systems. In this chapter, you will learn how to use Gephi and the data from *Draw Your First Network with Paper and Pencil,* on page 6 to build and layout a network of nutrients and food types, add attributes to its nodes and edges, and save the network as a presentation-quality image. You will also learn how to interchange network files between Gephi and NetworkX.

Worth 1,000 Words

Gephi[1] is not a part of NetworkX, and you must install it separately. Luckily, the installation process is straightforward.[2]

Here's a summary of Gephi's capabilities:

- Import existing networks in a variety of formats or create a new network.
- Edit a new or existing network by adding or removing nodes or edges.
- Change the size and color of node icons, the size and color of label font, and the color and thickness of edges based on node and edge attributes.

1. gephi.org
2. gephi.org/users/install/

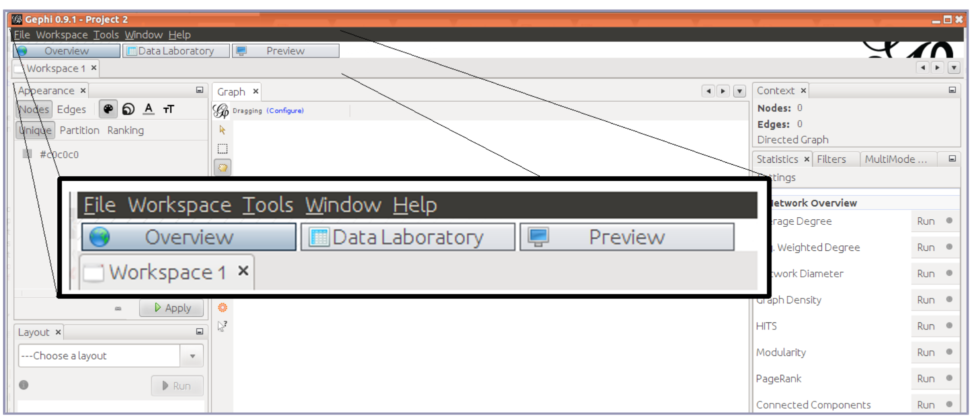
- Calculate basic network measures: centralities, clustering coefficients, path length distributions, connectedness, and modularity.
- Apply various layout engines to the network graph.
- Execute additional plug-ins.
- Export modified networks in a variety of formats.
- Save network visualization as a PNG, PDF, or SVG file.

If it looks like everything you ever wanted to do as a network analyst is already on the list, do not get overexcited. Gephi is an excellent network construction and analysis tool, but it is interactive. The human user is its slowest component (yes, this means you are slowing things down). Gephi cannot be programmed to execute batches of tedious analysis tasks, vary parameters automatically, or integrate with machine learning or predictive analytics software.

Nonetheless, Gephi is a great lightweight "Paintbrush"-style application and NetworkX companion. Let's use it to play with the nutrients network acquired in Chapter 3, *Introducing NetworkX*, on page 17.

Import and Modify a Simple Network with Gephi

The following figure shows the standard main window of Gephi without any loaded graphs.



The main window contains three tabs: Overview (for interactive network creation and exploration), Data Laboratory (for text and numeric editing and research), and Preview (for presentation-quality visualization). We will start in the Overview tab, then briefly stop at the Data Laboratory, and finish in the Preview.

The default Overview tab has four empty windows. The upper-left window, Appearance, is in charge of rendering. You will use it to control node and edge

presentation properties. The lower-left window, Layout, is behind the graph layout. The central window, Graph, shows the sketch of the network. The window on the right, with the tabs Statistics and Filters, is for network analysis and filtering (more about filtering on page 14). You can freely move the windows around, remove them, and add more windows from the Window menu.

To import an existing network file prepared by NetworkX or any other CNA software—in our case, nutrients.csv—open it with File > Open (Gephi displays only the file that it "knows" how to interpret.) Choose the Undirected graph type. Check if the number of detected edges and nodes makes sense. If the import is successful, you will see an ugly black-and-white sketch of your network in the Graph window. Don't worry: we will make it look awesome by the end of this chapter.

By default, Gephi treats networks as mathematical graphs and does not display node labels. Click the fat T button at the bottom of the Graph window, and the labels will show up, making the ugly sketch even uglier.

The original black color of the nodes is depressing. Before we go any further, click the artist's palette icon in the Appearance window, select the Unique tab, and choose a fun color—say, light blue—for all the nodes. (Don't forget to click Apply!)

Suppose that at this point you realize that eggs are a valuable source of selenium (they are) and you would like to add a connection between the respective nodes. You remember that eggs are already on the network, but you're not sure about selenium, so you add it to the network, just in case. (It is a mistake, but you will be able to correct it later.) Choose the Node Pencil tool in the Graph window, adjust the color and size of the new node, if desired, and click anywhere in the window. The position of the new node doesn't matter; the node will be moved around by the layout procedure.

Now, choose the Edge Pencil tool in the same window. Click the "eggs" node and then the new nameless one. Congratulations, you added a new edge!

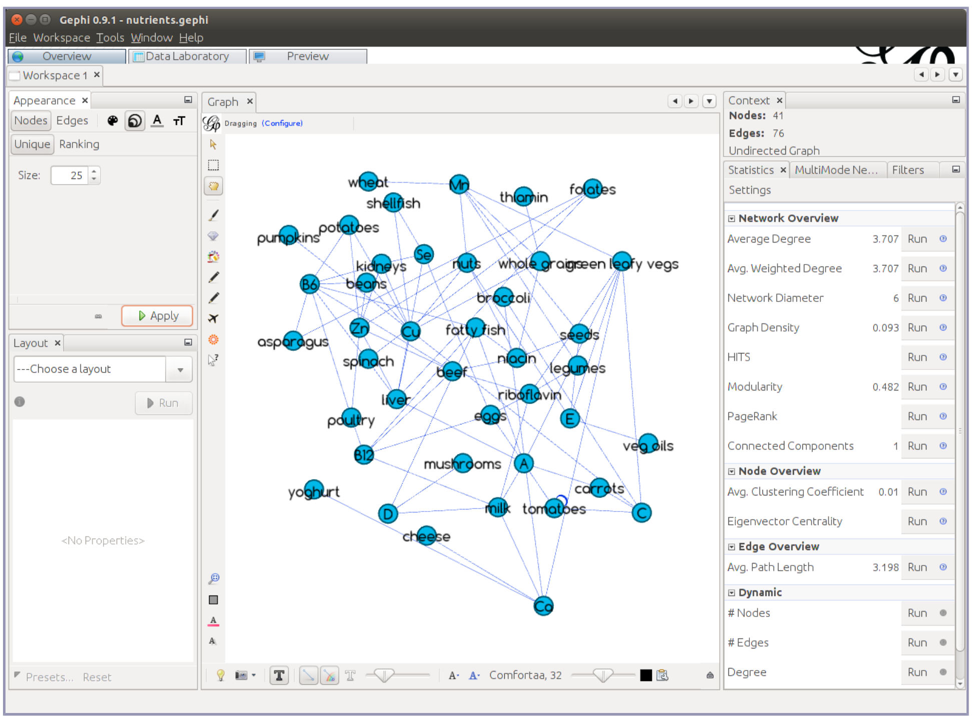
Finally, take care of the anonymous node. It deserves a name, so you're going to name it "Se" for "selenium." Click the Edit tab of the Appearance window, then the node of interest. Edit the label name (the default value is "<null value>")—and realize that now you have two selenium nodes!

Visit Data Laboratory

Data Laboratory (in addition to the Edit tab) is another place to look at the edges and nodes under a microscope and inspect and modify their properties. The

operation that you're looking for—detection of duplicate nodes—is hidden in the pull-down menu "More actions" at the top of the Data Table window. It is called "Detect and merge node duplicates." When you invoke the operation, Gephi reports that it found one pair of duplicates, and offers to merge them into one node. This procedure preserves all edges incident to the nodes. In particular, if there is an edge connecting the duplicates, it becomes a self-loop edge.

Unfortunately, Gephi does not tell you which nodes are duplicates. You can either trust the tool or manually find the duplicates in the table, select both nodes, open the pop-up menu, and select "Merge nodes…" in it. Either way, you can go back to the Overview window, and you'll see that there is now only one Se node, and it is properly connected to the eggs. The network now looks like the following figure. Node positions and fonts may differ, but surely the new image is a vast improvement over the black-and-white sketch.



Explore the Network

Network exploration in Gephi goes hand in hand with selecting visual properties. Let's paint and resize the graph nodes based on some of their measures.

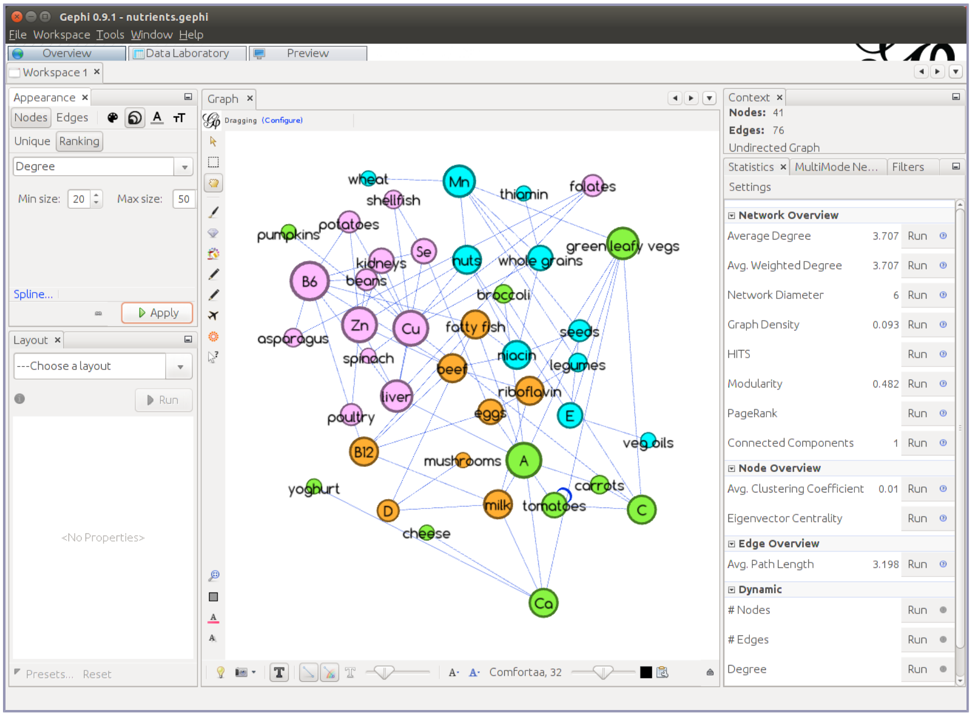
You will learn about network measures in Chapter 8, *Measuring Networks*, on page 83 and Chapter 11, *Unearthing the Network Structure*, on page 125. For now, it suffices to know several basic facts about two of them, as detailed in the following table:

Measure	Meaning
Degree	The number of immediate neighbors—adjacent nodes. The degree is a non-negative integer number. The larger the degree of a food item is, the more nutrients it provides. The larger the degree of a nutrient is, the more food items provide it.
Community structure	Nodes form tightly knit groups called communities. All foods and nutrients within a community serve some common purpose. Each community has a unique integer identifier called modularity class.

Node degree is the simplest possible node measure. There is no need to calculate it explicitly. To make node size proportional to the degree, click the icon with concentric circles in the Appearance window, then on the Ranking button. Select Degree from the "—Choose an attribute" pull-down menu. Select node sizes that correspond to the smallest and largest degrees (10 and 40 are good choices). And don't forget to click Apply. Can you see which nodes have the highest degrees?

Playing with size is fun; playing with color is more fun. Let's paint the nodes according to their modularity classes, as I've done in the figure on page 36. To partition a network into communities, click the Run button next to the Modularity command in the Network Overview section of the Statistics window. Proceed by clicking OK and Close in the next two dialogs. In the end, you will see a floating-point number next to Modularity. The number is a measure of the quality of the decomposition from *Outline Modularity-Based Communities*, on page 136. Return to the artist's palette icon, then click the Partition button. Select Modularity Class from the "—Choose an attribute" pull-down menu. Don't forget to click Apply. If you feel artistic, like me, play with the node colors. Painting the beef group pink and the vegetable group green is a no-brainer, but can you choose a good color for the vitamins?

If you plan to use Gephi for more sophisticated CNA jobs, the table on page 36 will help you find which sections of this book match the items in the Statistics window.

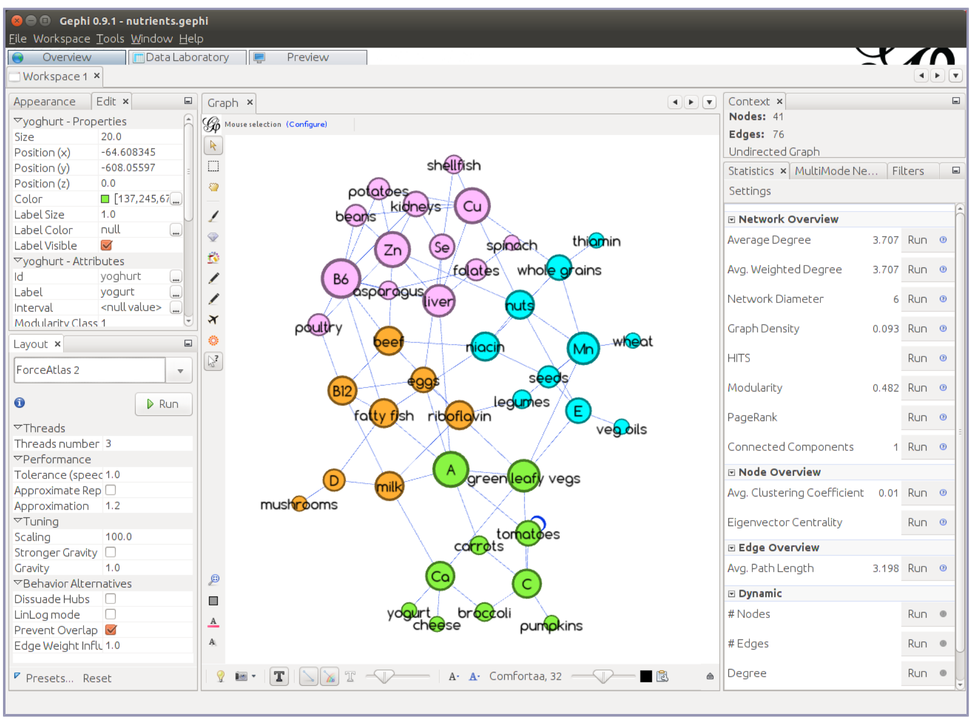


Measure	NetworkX reference
Average Degree	*Degree Centrality*, on page 92
Network Diameter	*Think in Terms of Paths*, on page 88
↑ Also calculates betweenness centrality	*Betweenness Centrality*, on page 93
↑ Also calculates closeness centrality	*Closeness and Harmonic Closeness Centrality*, on page 93
↑ Also calculates eccentricity	*Networks as Circles*, on page 91
Graph Density	*Start with Global Measures*, on page 83
HITS (Hubs and Authorities)	*HITS Hubs and Authorities*, on page 95
Modularity	*Outline Modularity-Based Communities*, on page 136
PageRank	*PageRank*, on page 94
Connected Components	*Split Networks into Connected Components*, on page 126
Avg. Clustering Coefficient	*Explore Neighborhoods*, on page 84
Eigenvector Centrality	*Eigenvector Centrality*, on page 94
Avg. Path Length	*Think in Terms of Paths*, on page 88

Sketch the Network

You're done with rough rendering, but the layout is still awful. Let's turn our attention to the lower-left corner of Gephi. Select your favorite layout from the "Choose a layout" pull-down menu. When a network is large (500 or more nodes), the Fruchterman-Reingold layout is usually the most efficient. For smaller networks, the ForceAtlas 2 layout, with some tweaking, works marvels. To make things easier, here's a tip: set the scaling to 100.0 (to place nodes reasonably far apart) and check the Prevent Overlap box. Run the tool for a while. You'll notice that after a couple of seconds, the nodes settle at their new positions, but the graph as a whole may continue drifting, rotating, or both.

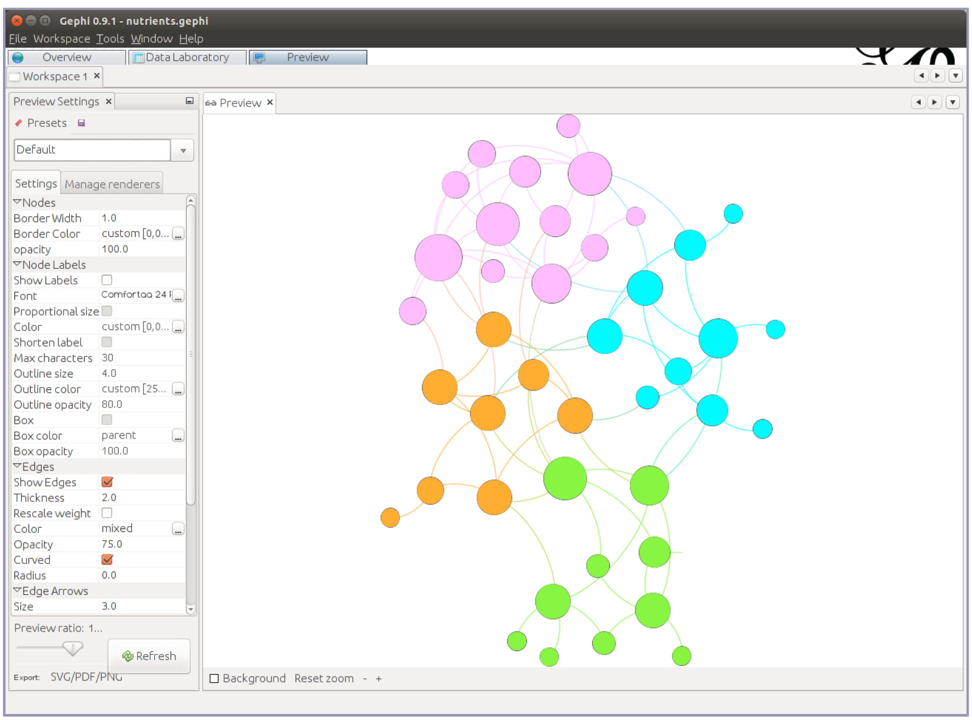
The last step is to adjust the labels, because surely some of them don't mind their manners and sit on top of each other. Select Label Adjust from the "Choose a layout" pull-down menu and run the tool for a couple of seconds. This layout engine distorts the original Fruchterman-Reingold but makes sure that neither nodes nor labels overlap. Hopefully, your network layout resembles the following figure.



The Graph window still shows a sketch, but this is a high-quality sketch that nicely displays the structure of the network of foods and nutrients. There are five compact groups in the network that could be tentatively called Veggies, Cereals, Meats, Proteins, and Folates. You can explore each group's internal composition, as well as connections to the other groups. You can even show this sketch to your boss or customer. But it would look much better when rendered at high resolution and converted to a presentation-quality image. (Save the project via File > Save As… into a .gephi file to avoid data loss if Gephi crashes!)

Prepare a Presentation-Quality Image

The final painting of the network takes place in the Preview tab. The tab has two windows: Preview Settings (to control the fine-level rendering engine) and Preview (to see the rendering results). When you switch to this tab, the Preview window will be empty. Click the Refresh button. Your network will be drawn using the default settings, with curved edges and no node labels (see the following figure). Don't worry—the labels have not been lost; they have been turned off.

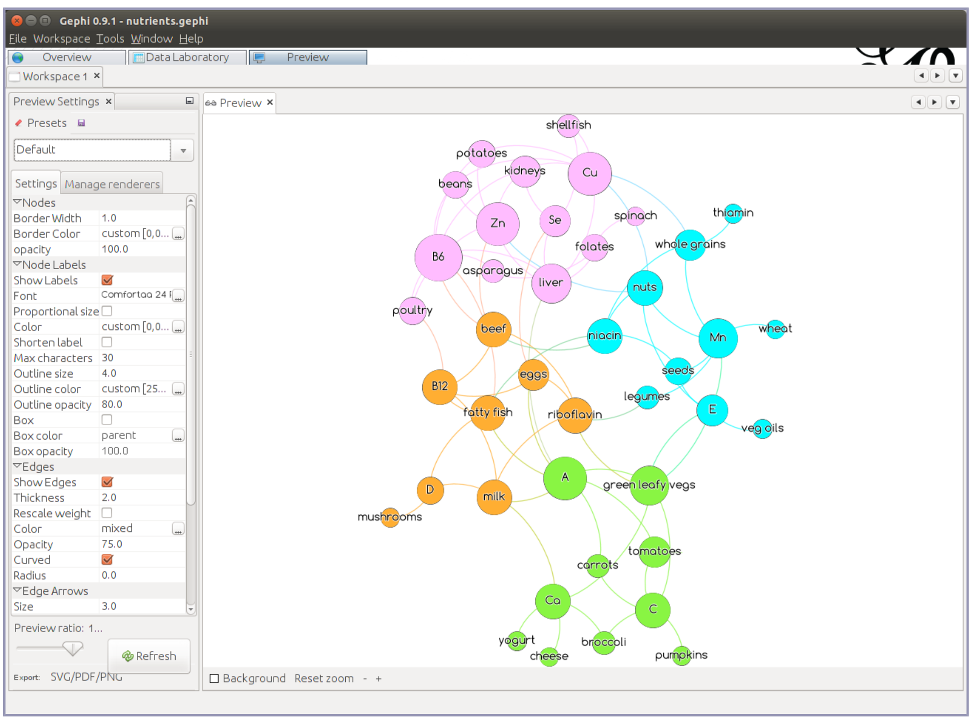


Gephi supports a variety of preset renderers. The default one is usually not the best one. I recommend using the "Text outline" preset configuration with some extra tweaking:

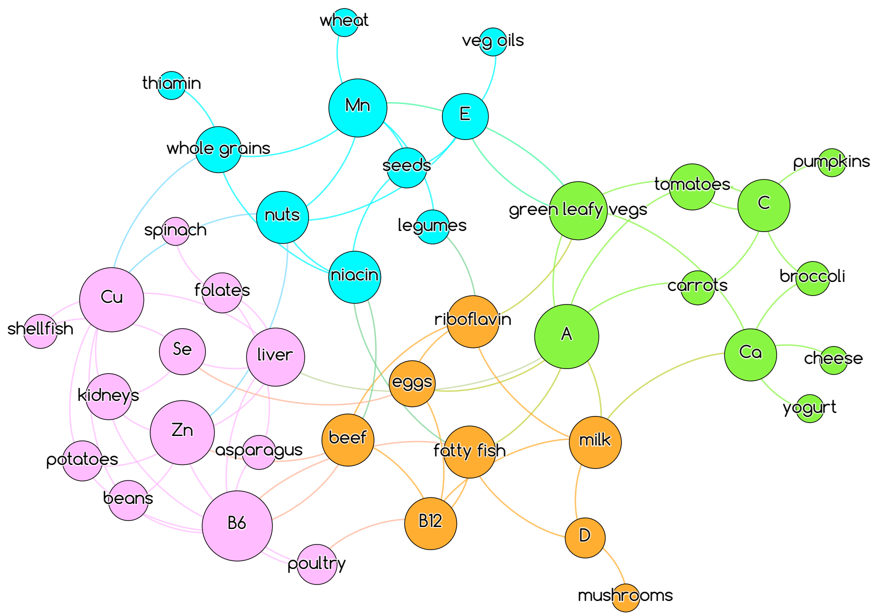
- Unclick "Proportional size."
- Increase the Font size (in this chapter, I used Comfortaa 24pt).
- Increase edge Thickness (I used 2.0).
- Set edge Opacity at 75.0.
- Optionally, unclick the Curved box.

Click the Refresh button again.

If you're not happy with what you see, you can make more changes. You can zoom into the network and zoom out of it, pan the image, control how many edges and nodes are rendered, and so on. You can go back and forth between the Preview and Overview to adjust layout and node and edge visualization properties. Eventually, your network will look similar to the following figure.



Once you like the rendering results, you can export the graph into a graphics file. Gephi provides exporters to SVG (editable vector format), PDF (editable vector format), and PNG (non-editable raster format). The final version of the network of foods and nutrients is in the figure on page 40.



A4 vs. Letter

The default page format for the Gephi PDF exporter is European A4 (also known as DIN A4 in Germany; it measures 210 by 297 millimeters, or 8.27” × 11.7”). A4 paper is slimmer and taller than the letter-size paper (at 8.5” × 11”) used in the United States, Canada, Chile, Colombia, Venezuela, the Philippines, and most Central American countries. If you live in one of the “letter” countries, make sure to change the paper size.

Once again, you are invited to compare this figure with the hand-drawn network on page 8!

Combine Gephi and NetworkX

There is no implicit integration between Gephi and NetworkX. However, you can use Gephi graph file exporter (File > Export > Graph file…) to save your network into a file that could be imported by NetworkX. GraphML is the preferred interchange format because it preserves all calculated measures (such as centralities and modularity classes) as node attributes. This way you could use Gephi to perform a quick-and-dirty interactive analysis of a network and save it into a .graphml file for further processing.

You just learned one more way to build and analyze networks by hand, and you now understand that analyzing one not-so-large, pre-packaged complex network with Gephi is a pleasure. You also understand that constructing and analyzing many large networks by hand is hard, error-prone, and most of the time infeasible. You are now ready for the first full-Python case study.

Looking down, he surveyed the rest of his clothes, which in parts resembled the child's definition of a net as a lot of holes tied together with string…

➤ *Morley Roberts, English novelist and short story writer*

CHAPTER 5

Case Study: Constructing a Network of Wikipedia Pages

So far you have learned two ways of constructing a complex network: a hard one (from a CSV file and further in Gephi, *Construct a Simple Network with NetworkX,* on page 17) and a very hard one (by hand on paper, *Draw Your First Network with Paper and Pencil,* on page 6). What is hard for small networks may be impossible for medium-to-large scale networks; it may be impossible even for small networks if you must repeat the analysis many times. The case study in this chapter shows you how to construct a large network in an easy way: by automatically collecting node and edge data from the Internet.

The other goal of this study (aside from mastering new network construction techniques) is quite pragmatic. Wouldn't you want to know where the complex network analysis fits in the context of other subjects and disciplines? An answer to this question is near at hand: on Wikipedia.[1]

Let's start with the Wikipedia page about complex networks—the seed page. (Unfortunately, there is no page on complex network analysis itself.) The page body has external links and links to other Wikipedia pages. Those other pages presumably are somewhat related to complex networks, or else why would the Wikipedia editors provide them?

To build a network out of the seed page and other relevant pages, let's treat the pages (and the respective Wikipedia subjects) as the network nodes and the links between the pages as the network edges. You will use snowball sampling (explained on page 7) to discover all the nodes and edges of interest.

1. en.wikipedia.org/wiki/Complex_network

As a result, you will have a network of all pages related to complex networks and hopefully, you will make some conclusions about it.

Get the Data, Build the Network

The first half of the project script consists of the initialization prologue and a heavy-duty loop that retrieves the Wikipedia pages and simultaneously builds the network of nodes and edges.

This section uses Wikipedia.

Let's first import all necessary modules. We will need the module wikipedia for fetching and exploring Wikipedia pages, the operator itemgetter for sorting a list of tuples, and, naturally, networkx itself.

To target the snowballing process, define the constant SEED, the name of the starting page. As a side note, by changing the name of the seed page, you can apply this analysis to any other subject on Wikipedia.

Last but not least, when you start the snowballing, you will eventually (and quite soon) bump into the pages describing ISBN and ISSN numbers, the arXiv, PubMed, and the like. Almost all other Wikipedia pages refer to one or more of those pages. This hyper-connectedness transforms any network into a collection of almost perfect gigantic stars, making all Wikipedia-based networks look similar. To avoid the stardom syndrome, treat the known "star" pages as stop words in information retrieval—in other words, ignore any links to them. Constructing the black list of stop words, STOPS, is a matter of trial and error. I put twelve subjects on it; you may want to add more when you come across other "stars." I also excluded pages whose names begin with *"List of"*, because they are simply lists of other subjects.

wiki2net.py
```
from operator import itemgetter
import networkx as nx
import wikipedia

SEED = "Complex network".title()

STOPS = ("International Standard Serial Number",
         "International Standard Book Number",
         "National Diet Library",
         "International Standard Name Identifier",
         "International Standard Book Number (Identifier)",
         "Pubmed Identifier", "Pubmed Central",
         "Digital Object Identifier", "Arxiv",
         "Proc Natl Acad Sci Usa", "Bibcode",
         "Library Of Congress Control Number", "Jstor")
```

The next code fragment deals with setting up the snowballing process. A breadth-first search, or BFS (sometimes known to computer programmers as a snowballing algorithm), must remember which pages have been already processed and which have been discovered but not yet processed. The former are stored in the set done_set; the latter, in the list todo_lst and set todo_set. You need two data structures for the unprocessed pages because you want to know whether a page has been already recorded (an unordered lookup) and which page is the next to be processed (an ordered lookup). The Aside titled “Ready, Set(), Go” on page 25 explains why the two operations favor different data structures.

Snowballing an extensive network—and Wikipedia with 5,452,810 articles in the English segment alone can produce a huge network!—takes considerable time. Suppose you start with one seed node, and let’s say it has N≈100 neighbors. Each of them has N neighbors, too, to the total of ≈N+N×N nodes. The third round of discovery adds ≈N×N×N more nodes. The time to shave each next layer of nodes grows exponentially. For this exercise, let’s process only the seed node itself and its immediate neighbors (layers 0 and 1). Processing layer 2 is still feasible, but layer 3 requires N×N×N×N≈10^8 page downloads—close to one year of your machine time. To keep track of the distance from the currently processed node to the seed, store both the layer to which a node belongs and the node name together as a tuple on the todo_lst list.

wiki2net.py

```
todo_lst = [(0, SEED)] # The SEED is in the layer 0
todo_set = set(SEED)   # The SEED itself
done_set = set()       # Nothing is done yet
```

The output of the exercise is a NetworkX graph. The next fragment will create an empty directed graph that will later absorb discovered nodes and edges. We choose a directed graph because the edges that represent HTML links are naturally directed: a link from page A to page B does not imply a reciprocal link.

The same fragment primes the algorithm by extracting the first “to-do” item (both its layer and page name) from the namesake list.

wiki2net.py

```
F = nx.DiGraph()
layer, page = todo_lst[0]
```

It may take a fraction of a second to execute the first five lines of the script. It may take the whole next year or longer to finish the next twenty lines because they contain the main collection/construction loop of the project.

wiki2net.py
```
while layer < 2:
    del todo_lst[0]
    done_set.add(page)
    print(layer, page) # Show progress

    try:
        wiki = wikipedia.page(page)
    except:
        layer, page = todo_lst[0]
        print("Could not load", page)
        continue

    for link in wiki.links:
        link = link.title()
        if link not in STOPS and not link.startswith("List Of"):
            if link not in todo_set and link not in done_set:
                todo_lst.append((layer + 1, link))
                todo_set.add(link)
            F.add_edge(page, link)

    layer, page = todo_lst[0]
print("{} nodes, {} edges".format(len(F), nx.number_of_edges(F)))
# 11597 nodes, 21331 edges
```

The loop is programmed to collect all nodes that are at most two steps away from the seed node. They are reachable from the nodes in layer 1, and all those nodes will have been harvested when the loop terminates. The loop body consists of the following four blocks:

1. Remove the name page of the current page from the todo_lst, and add it to the set of processed pages. If the script encounters this page again, it will skip over it.

2. Attempt to download the selected page. If the attempt is unsuccessful (things happen!), proceed to the next page from the "to-do" list.

3. Evaluate each link. If the subject is not blacklisted and not a list itself, the script adds an edge to the graph between the current node and the linked page. If the script did not process the linked page before and it is not on the "to-do" list, add it to the list and corresponding set. Note that the highlighted code line is involved in the network construction—the only line in the script!

4. Take the next page name from the "to-do" list. Hopefully, the list is not empty. If it is—congratulations, you just downloaded the complete Wikipedia!

The network of interest is now in the variable F. But it is "dirty": inaccurate, incomplete, and erroneous.

Eliminate Duplicates

Many Wikipedia pages exist under two or more names. For example, there are pages about *Complex Network* and *Complex Networks*. The latter redirects to the former, but NetworkX does not know about the redirection.

Accurately merging all duplicate nodes involves natural language processing (NLP) tools that are outside of the scope of this book. It may suffice to join only those nodes that differ by the presence/absence of the letter *s* at the end or a hyphen in the middle.

Start removing self-loops (pages referring to themselves). The loops don't change the network properties but affect the correctness of duplicate node elimination.

Now, you need a list of at least *some* duplicate nodes. You can build it by looking at each node in `F` and checking if a node with the same name, but with an *s* at the end, is also in `F`. Pass each pair of duplicated node names to the function `nx.contracted_nodes(F,u,v)` that merges node `v` into node `u` in the graph `F`. The function reassigns all edges previously incident to `v`, to `u`. If you don't pass the option `self_loops=False`, the function converts an edge from `v` to `u` (if any) to a self-loop.

wiki2net.py

```
F.remove_edges_from(F.selfloop_edges())
duplicates = [(node, node + "s") for node in F if node + "s" in F]
for dup in duplicates:
    F = nx.contracted_nodes(F, *dup, self_loops=False)
duplicates = [(x, y) for x, y
              in [(node, node.replace("-", " ")) for node in F]
              if x != y and y in F]
for dup in duplicates:
    F = nx.contracted_nodes(F, *dup, self_loops=False)
nx.set_node_attributes(F, "contraction", 0)
```

Thou Shall Not Contract Self-Loops

Due to a bug in the implementation of `nx.contracted_nodes()` in NetworkX 1.11 and earlier versions, the function *fails* to merge duplicates if one of them has a self-loop. Hopefully, this bug will be fixed in the future. For now, either permanently delete all self-loop edges before eliminating duplicates, or compose a list of the self-loop edges, remove them, eliminate the duplicates, and add the self-loop edges back to the network graph.

As a side effect, `nx.contracted_nodes()` creates a new node attribute (see *Add Attributes, on page 23*) called *contraction*. The value of the attribute is a dictionary, but GraphML does not support dictionary attributes. The highlighted

code line sets the *contraction* property to 0 for all nodes to avoid further troubles with exporting the graph. You could also delete the attribute for each node `n` with `del n["contraction"]` in a loop.

Truncate the Network

Why did you go through all those Wikipedia troubles? First, to construct a network of subjects related to complex networks—and here it is. Second, to find other significant topics related to complex networks. But what is the measure of significance?

You will discover a variety of network measures in Chapter 8, *Measuring Networks*, on page 83. Some of them are excellent proxies for node importance. For now, let's concentrate on a node *indegree*—the number of edges directed into the node. (In the same spirit, the number of edges directed out of the node is called *outdegree*.) The indegree of a node equals the number of HTML links pointing to the respective page. If a page has a lot of links to it, the topic of the page must be significant.

The choice of indegree as a yardstick of significance incidentally makes it possible to shrink the graph size by almost 75 percent. The extracted graph has 11,390 nodes and 20,392 edges—an average of 1.8 edges per node. Most of the nodes have only one connection. (Interestingly, there are no isolated nodes with no connection in the graph. Even if they exist, you will not find them because of the way snowballing works.) You can remove all nodes with only one incident edge to make the network more compact and less hairy without hurting the final results. Why?

- If a node has one incoming edge, then removing the node affects the outdegree of some other node, but you do not care about outdegrees.
- If a node has one outgoing edge (and the node is not the seed), you could not have found it, at least not with snowballing.

As you can see, the following code fragment safely removes 75 percent of the nodes and 45 percent of the edges, raising the average number of edges per node to 3.9.

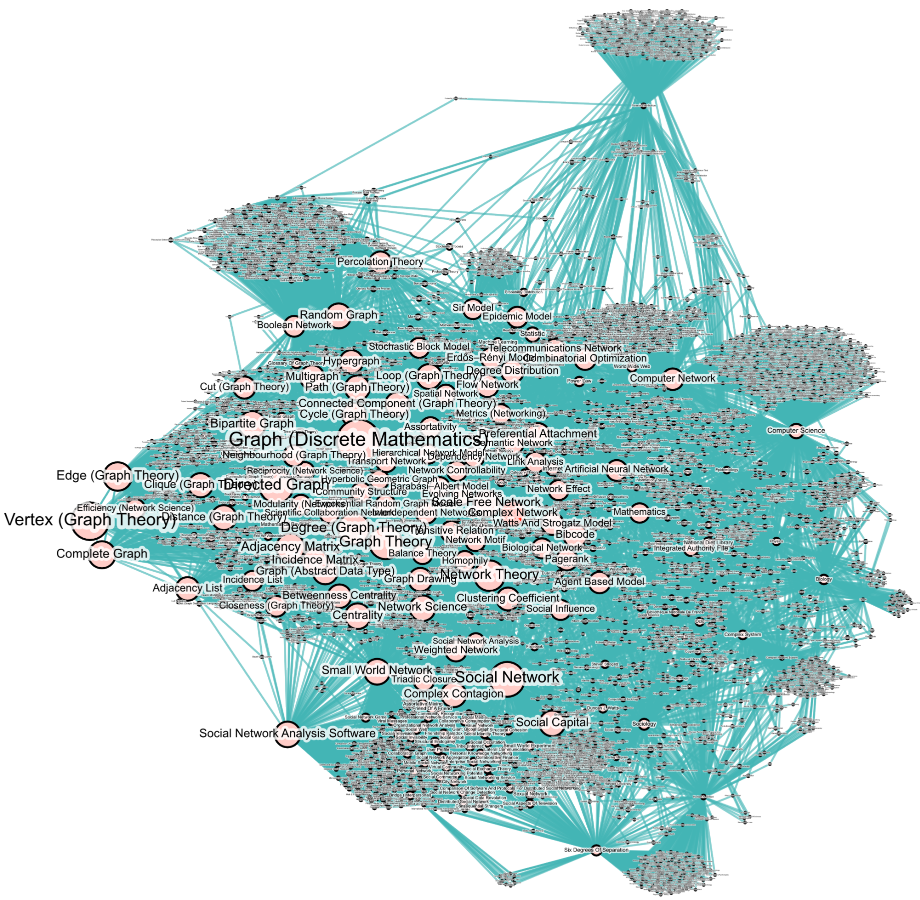
wiki2net.py
```
core = [node for node, deg in F.degree().items() if deg >= 2]
G = nx.subgraph(F, core)
print("{} nodes, {} edges".format(len(G), nx.number_of_edges(G)))
# 2995 nodes, 11817 edges
nx.write_graphml(G, "cna.graphml")
```

Start by calling `F.degree()`. The method returns a dictionary with nodes as keys and degree as values. Note that at this point, you don't need to distinguish indegrees and outdegrees (for the reasons explained previously). Expand the dictionary into a list of key/value tuples and select the nodes whose degree is at least 2—the "dense core" of the network.

Function `nx.subgraph(F, core)` collects all `core` nodes from `F` and all edges connecting them and builds a new graph `G`—a subgraph of `F`. (Naturally, `F` has a lot of different subgraphs. Even `F` itself is a subgraph of `F`.) `G` is a truncated version of `F`. Write it to a GraphML file so that you don't have to rebuild it if you need it later again.

Explore the Network

The following figure is a `Gephi` rendering of `G`. The "Complex Network" node is barely visible right in the middle of the image denoted with a darker color. Node and label font sizes represent the indegrees. The most in-connected, most significant nodes are in the upper-left corner of the network. What are they?



The last code fragment of the exercise efficiently calculates the answer by calling the method G.in_degree(). The method (and its sister method G.out_degree()) are very similar to G.degree() except that they report different edge counts.

wiki2net.py
```
top_indegree = sorted(G.in_degree().items(),
                      reverse=True, key=itemgetter(1))[:100]
print("\n".join(map(lambda t: "{} {}".format(*reversed(t)), top_indegree)))
```

Sort the list of item tuples in the order of decreasing indegrees, pick the top one hundred items, and print their indegrees and names. These are the one hundred most significant subjects that, according to Wikipedia, go along with complex networks. The first twenty-five of them are listed in the table on page 49.

It appears almost magical that this book covers the majority of these and the remaining seventy-five automatically extracted most significant topics. On second thought, the outcome of the experiment merely confirms that the Wikipedia link structure reflects the structure of the complex networks analysis field. I encourage you to use the code from this case study to explore other Wikipedia subjects that may be of interest to you.

This chapter presented a complete complex network construction case study, starting from the raw data in the form of HTML pages, all the way to an analyzable annotated network graph and a simple exploratory exercise. This is a good foundation for more systematic complex network studies.

In the Next Part

You are about to move away from simple networks of nutrients and Wikipedia pages to the vast and venerable realm of social networks. In fact, as you learned in the introduction, social network analysis predates complex network analysis and serves as one of the cornerstones of the CNA. In the next part, you will learn how to construct, measure, interpret, and understand complex social networks, as well as the networks that resemble them.

Indegree	Subject	Covered in this book?
61	Graph (Discrete Mathematics)	✓
54	Vertex (Graph Theory)	✓
49	Directed Graph	✓
48	Social Network	✓
44	Graph Theory	Somewhat
44	Degree (Graph Theory)	✓
44	Network Theory	
39	Edge (Graph Theory)	✓
39	Adjacency Matrix	✓
39	Complete Graph	✓
38	Bipartite Graph	✓
37	Scale Free Network	✓
37	Graph (Abstract Data Type)	
36	Social Capital	
36	Network Science	
36	Small World Network	✓
36	Incidence Matrix	✓
36	Social Network Analysis Software	Somewhat
36	Centrality	✓
36	Loop (Graph Theory)	✓
35	Complex Contagion	
35	Complex Network	✓
35	Random Graph	✓
35	Path (Graph Theory)	✓
35	Distance (Graph Theory)	✓
35	Graph Drawing	✓

Part II

Networks Based on Explicit Relationships

Social networks—the networks of individuals, groups, or organizations—are historically the longest-studied complex networks. They are based on explicit relationships between the nodes, such as friendship, kinship, membership, and interaction.

If dogs could reason and criticize us they'd be sure to find just as much that would be funny to them, if not far more, in the social relations of men, their masters—far more, indeed.

➤ *Fyodor Dostoyevsky, Russian writer*

CHAPTER 6

Understanding Social Networks

Individuals, groups, and organizations also form networks. Such networks are called social networks. They are historically the longest-studied and probably the most familiar and intuitive complex networks. Social network nodes are explicitly related through friendship, kinship, and membership.

In this chapter, you will learn the taxonomy of social networks and their edges. You will understand the role of weak and strong edges in information dissemination and preservation, and the importance of centrality measures. In the end, you will have a glance at synthetic networks and learn why one needs them.

Understand Egocentric and Sociocentric Networks

Personal social networks are complex networks of persons or social animals. (No, whales and elephants do not have their own Facebook, but if you are intrigued, look at *Animal Social Networks [KJFC15]*!) Respectively, nodes represent people, and edges represent significant social relationships between people: kinship (remember the family tree on page 3?), friendship, acquaintanceship, subordination, and the like. Some of the relationships are typically directed (subordination, some subtypes of kinship), and others are undirected (friendship, acquaintanceship), giving rise to the namesake graphs. They may have different weight (*Distinguish Strong and Weak Ties,* on page 66), leading to weighted graphs. One can include any or all of these relationships in one network, ending up with multigraphs and other pseudographs. A social network is truly a complex one!

And the simplest form of a complex social network is an egocentric network.

Egocentric Networks

An egocentric network (or ego network, for short) is the social network of a particular individual. An ego network includes all the individual's contacts and all the relationships among them. Using the terminology from Chapter 5, *Case Study: Constructing a Network of Wikipedia Pages*, on page 41, an ego network includes the nodes from level 1, and the network of subjects related to complex networks is the ego network of the subject "complex networks."

Egocentric networks are used to understand the structure, function, and composition of connections around a single person. Unlike sociocentric networks, they are bounded and focus on individuals (rather than groups).

The central node of an ego network is referred to as *ego* (as in *egoism* and *alter ego*); all the other nodes are called *alter* (as in *alternative* and *alter ego*, again).

To construct a social ego network, start with an ego—say, yourself. Obtain the list of the ego's contacts—the alters. If you explore a social networking website, the list of alters is often called "friends list," "list of subscribers," or "list of followers." You can download it by using the site API, by scraping and parsing the site's HTML code, or, if nothing else works, by copying and pasting the data by hand.

When a Social Network Is Not a Social Network

When your friends say "social network," chances are they are using the words wrong. For example, Facebook is not a social network.

Facebook is a social networking website (SNS)—a website that facilitates social networking by augmenting traditional offline, face-to-face communications with instant online communications. The difference between a social network and a social networking website is like the difference between club members and the club building: while it is easier for the club members to meet in the club building, the building is not strictly necessary for the club to function.

Your mom is still your mom and belongs to your ego network, whether she is on Facebook or not.

If you are a sociologist, anthropologist, or another researcher in the field of social sciences, you may need to deal with real people rather than digital lists. Your principle inquiry tool is probably a name generator (they are described in detail in *Social Network Analysis [KY08]* and other SNA-related books). A name generator is a list of contacts—alters—prepared on your request by the

ego person—the person who will be the center of the network. (If you're working on your ego network, you have to make the list yourself.) Often name generators have restricted length to facilitate recollection, but ego networks derived from shortened lists have less elaborate structure.

One way or another, digitally or by hand, you will get a list of some or all alters. You can arrange them into a star network with the ego at the center because all of them are connected to the ego. But that's not enough. Now you have to repeat the contact collection procedure for each of the alters: either by calling the APIs/scraping/copying/pasting or by soliciting names through name generator surveys. With the median number of friends on Facebook being between 155 and 500, depending on whom you trust, the process of data collection may become quite daunting, unless properly automated. One may only wonder how people researched ego networks before Facebook. (Hint: Before Facebook, there was MySpace.[1])

Ego network construction significantly differs from the snowballing process on page 7 in the way you treat newly discovered nodes. An ego network does not extend beyond the alters. You're supposed to discard any detected node that is not an alter (which is inefficient, but you can save the unwanted nodes for the future—say, for a full social network analysis).

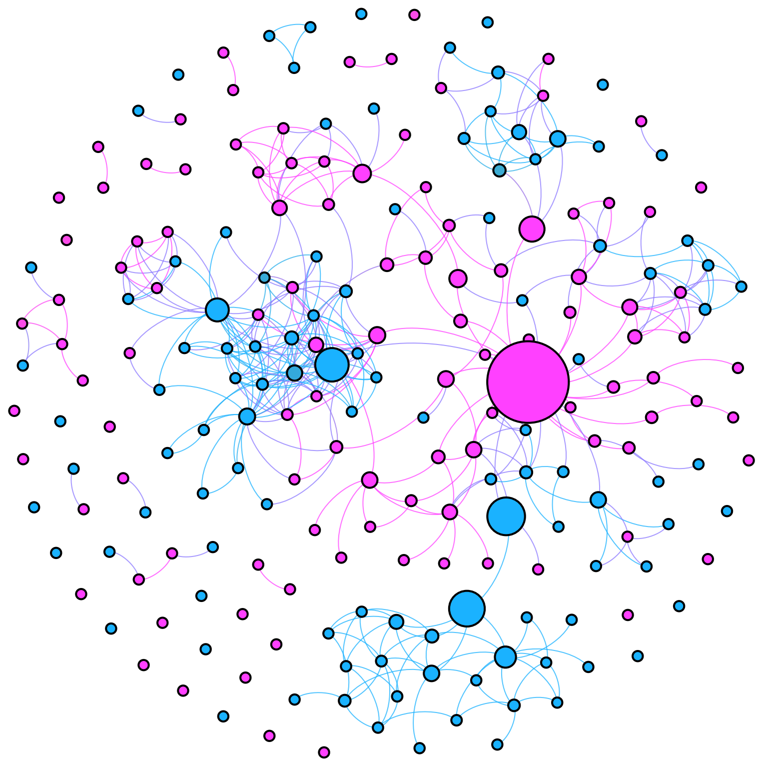
You Could Have Had Your Facebook Ego Network

The official Facebook application programming interface (API) v1.0 allowed Facebook apps to download your friends' friend lists. That's all you needed to construct your ego network programmatically. Lada Adamic, a prominent complex network researcher, wrote a program called `GetNet` that used the Facebook APIs to build ego networks. The program worked well until May 1, 2015, when Facebook retired v1.0 due to privacy concerns. Luckily, I collected my Facebook ego network back in 2013 (see the figure on page 56). It is four years old but better old than nothing.

Once you harvest the ego and all the alters, remove the ego from the network. It is the center of the giant star with the highest connectivity. It dominates the network. It is the tree that makes it hard to see the forest. Removing the ego does not cause any information loss: if needed, just put it back and connect to each existing node.

As an example of a real ego network, let's have a look at my Facebook ego network constructed in 2013. The graph, anonymized for privacy reasons, is shown in the figure on page 56.

1. myspace.com



Here are some facts about it:

- Each node represents one of my friends, relatives, colleagues or acquaintances.
- Each edge represents what Facebook calls a “friendship,” but in reality can stand for anything from “she is my sweetheart” to “what’s-his-name.”
- The network does not have “my” node because each shown node is implicitly connected to me.
- Some nodes look completely isolated. They are not: each node is connected to me, and there may be other contacts to the nodes that are not my direct contacts.
- Pink and blue nodes represent female and male contacts.
- Some nodes congregate and form dense subgraphs. Some of these subgraphs are all-male, some are almost all-female, and some have mixed gender population. The subgraphs represent different aspects of my social

life: family, close friend circle, current and past jobs, and hobbies. An ego network is like a spectroscope that separates your alters into a social spectrum.

- Node size represents betweenness centrality (a measure of node importance; see *Betweenness Centrality*, on page 93).

  This fact goes with a tricky question: which node most likely represents my spouse?

Most of these facts are true about most of the human ego networks, but beware: an ego network is only a subgraph of a bigger social network. Anything you measure in an ego network—diameter, centralities, clustering coefficients (Chapter 8, *Measuring Networks*, on page 83)—is an approximation of the same measure for the same node in the bigger graph. Let's next have a look at sociocentric networks, where everyone is an ego and alter at the same time.

Sociocentric Networks

A sociocentric network, or just a social network, is any social network that is not egocentric. Ideally, a sociocentric network is a combination of the ego networks of all egos and includes all relevant (whatever it means to you as a researcher) alters. For example, a social network of all active Facebook users includes all ≈2.01 billion nodes representing active Facebook members (in 2017) and ≈0.25 trillion edges representing their friendships.[2] A complete social network of all living human beings has ≈7.44 billion nodes; the number of edges must be no more than 0.66 trillion if we believe in *Dunbar's number [Dun98]*—150, the number of individuals with whom a person can maintain stable relationships.

A sociocentric network is the prime focus of attention of social network analysts. It reveals all significant relationships of each actor in the network, exposes hierarchical groups of actors, and provides a framework for explaining the structure and evolution of individual edges and node groups.

A non-trivial social network, regardless of its size, is a complex network. What makes it distinctive is not the size but the interpretation: the social theories that stand behind the degree distributions, centralities, local network topology, community structure, and network evolution. The table on page 58 lists some examples of possible social interpretation of complex network properties. Some of them will be covered in this book. In this table, I call nodes "actors" to emphasize their human nature.

2. bigthink.com/praxis/do-you-have-too-many-facebook-friends

Network property	Examples of social interpretation
Local topology	Structural equivalence: if two actors have similar connections to other actors, they are similar or equivalent. Triadic closure: two friends of an actor eventually become friends. Balance theory: a friend of friend is a friend, a friend of a foe is a foe, and so on.
Degree and eigenvector centrality	Social capital: an actor produces common good for the friends. Influence: an actor causes a change in behavior in the friends.
Closeness centrality	Influence: see above. Information dissemination/diffusion: how good are actors in broadcasting or sharing information?
Betweenness centrality	Information dissemination: see above. Brokerage: how good are actors in serving as "go-betweens"?
Community structure	Homophily (cognitive balance): "birds of a feather flock together." Knowledge preservation: actors in tightly knit communities preserve knowledge. Complex contagion: a gang of interconnected infected actors is a source of contagion.
Degree distribution	Small world (six degrees of separation): any two actors on average are connected by six "handshakes." Friendship paradox: "my friends have more friends than I do."
Network dynamics	Preferential attachment (Pareto principle): "the [actors] rich [in friends] get richer."

The table is not complete by any means, but it gives you a sample of social research questions and SNA/CNA machinery typically associated with them. If interested, see *Social Network Analysis: A Handbook [Sco00]* and *Exploratory Social Network Analysis with Pajek (Structural Analysis in the Social Sciences) [NMB11]* for a comprehensive coverage of social network analysis with the emphasis on the *social* aspects.

Acquisition of Social Networks

Practically, you almost never get a complete social network of interest for several reasons.

1. Real social networks, especially those implemented through social networking websites (see the Aside titled "When a Social Network Is Not a Social Network" on page 54), are often huge. Obtaining them may take more time than you can afford.

2. Real social networks are dynamic. Nodes and edges are added and removed as you construct the network. By the time you fetch the last node and edge, the rest of the graph may already be out of sync with the real network.

However, you can still get a decent approximation of a social network using either snowball sampling or random node sampling.

The principal difference between two sampling approaches is the source of the nodes to process. A snowballing algorithm maintains a list of nodes that have been discovered but not visited yet (in other words, the nodes "at the other end" of the edges incident to the already visited nodes). The algorithm adds newly discovered nodes to the list and removes the discovered nodes after visiting them.

A random sampling algorithm needs an exogenous list of the seed nodes, like:

1. Actors interested in a particular event or subject (say, the 122,079,386 Facebook users who like the page of Cristiano Ronaldo, a Portuguese-born soccer megastar)

2. SNS users who are currently online (as once implemented on *Odnoklassniki*,[3] the second-largest Russian language SNS)

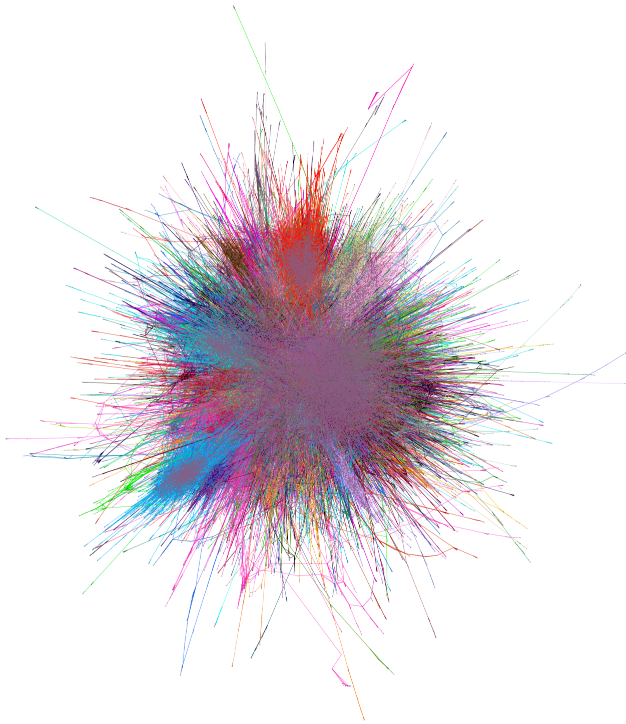
3. Persons with randomly chosen numerical SNS user IDs (say, 112424342928081542774 on Google+[4])

4. Persons randomly chosen from a telephone book

Randomness is the key to successful sampling. If the choice of the seed nodes is not random, then a sampling algorithm may go astray and collect an unrepresentative part of the network. Frankly speaking, examples 1 and 2 from the previous list are not entirely random. The first is biased toward the actors with particular interests; the second favors the population in a given time zone and with a particular online activity pattern.

3. ok.ru
4. plus.google.com

Once you have a list of seed nodes, you can obtain their ego network graphs, combine them into one large graph, and enjoy a sampled sociocentered network. As an example of what you may end up with, the following figure shows 165,795 nodes and 434,118 edges of MoiKrug[5] (a Russian-language counterpart of LinkedIn[6]) that I collected in 2009. Different colors in the figure denote different tight network neighborhoods.



And just to remind you: what you see in the figure is 0.012 percent of the current Facebook population. Some social networks are complex by all measures.

Signed Networks

Some social (and not only social) networks belong to a class of signed (as opposed to unsigned) networks. There is not much special about them—except that they are weighted, and the weights can take negative values. This feature allows using the same type of ties to represent both positive and negative aspects of relationships. For example, a negatively weighed friendship tie between Alice and Chuck may signify that the folks are foes rather than friends.

5. moikrug.ru
6. www.linkedin.com

Signed networks are dangerous because their visual inspection does not reveal the true meaning of signed ties. Any network analysis algorithm that disregards weights would be fooled into believing that a tie is an indicator of proximity, while in the case of Alice and Chuck, it is just the opposite.

Nonetheless, some social theories (including the balance theory mentioned in the table on page 58) make heavy use of signedness. As a professional, you should be ready to handle it.

Prepared Social Networks

If you're not in the mood to crawl a social network yourself or need a typical (but not any particular) network for your experiment, you have two more options. You can either generate a synthetic network with preset properties (*Appreciate Synthetic Networks*, on page 63) or download an empiric network from the Stanford Large Network Dataset Collection compiled by Jure Leskovec and Andrej Krevl in 2014.[7]

The collection provides free access to ninety snapshots of various complex networks grouped into seventeen categories. At least thirty-three of them describe social and communication networks obtained from Facebook, Google+, Twitter, LiveJournal, Slashdot,[8] and similar sites. We will use one of the networks from the collection—Enron email communication network—in the next section.

Another source of publicly available empiric networks is the Koblenz Network Collection (KONECT) by Jérôme Kunegis, featuring 261 datasets.[9]

Recognize Communication Networks

For most of the world, a communication network is an electrical (digital or analog) circuitry that connects terminal communication equipment, such as wired and cellular phones, computers, modems, TV sets and cable boxes, and the like. Not so in social network analysis.

A social communication network is a social network where the edges represent a communication relation: *"channels through which messages may be transmitted" [KY08]*. In other words, two actors are adjacent if they directly communicate or have a propensity for direct communication. The communication medium is not of our concern: it could be face-to-face, verbal over a

7. snap.stanford.edu/data/
8. slashdot.org
9. konect.uni-koblenz.de

phone, email, Internet chat, prison tap code, paper “snail mail,” or even carrier pigeons. As long as an archetypal Alice can reliably send a message to an archetypal Bob, anything goes. (If you haven’t met Alice and Bob yet, meet them in *The Tale of Alice, Bob, and Chuck*, on page 62.)

Communication networks still connect people or social animals, but the connectedness is derived not from a relatively stable relationship (friendship, kinship, or membership), but from short, often instantaneous interactions. The interactions may be one-way or two-way; frequent, occasional, or even isolated. It is up to you to decide what communication pattern constitutes an edge. (I will help you in *Distinguish Strong and Weak Ties*, on page 66.)

The most friendly communication medium is corporate emails. In a case of major conundrums, corporate email archives may become public. The largest public corpus is the Enron email communication network published by the Federal Energy Regulatory Commission during the investigation of the company’s criminal activity in 2008.[10] The network has 36,692 nodes corresponding to 150 Enron employees and other addressees, and 183,831 edges—direct node-to-node email communications. The edges (and the network) are directed. An edge connects the sender and the recipient. The researchers who constructed the communication network did not incorporate the intensity of interactions in the published dataset. We cannot tell which edges represent strong and weak message exchanges.

There are some unique issues associated with communication networks. First, some communication media allow information broadcast. For example, a speaker at a convention addresses all attendees at once. An email message can be sent to many recipients. A public Internet forum post is usually read by more than one reader. Assuming that each attendee/reader/recipient is not a random passerby, you can model group communications as a star by connecting the speaker/poster/sender to each addressee with a separate edge.

The Tale of Alice, Bob, and Chuck

Alice and Bob are fictional characters frequently used in cryptology and communication theory to represent communicating parties. They were invented in 1978 and lived happily ever after. Alice usually is the sender: she initiates the conversation by sending a message to Bob, who is the recipient. Sometimes, the communication is marred by Chuck the Bad Guy whose goal is to intercept, interrupt, modify, or fabricate the message.

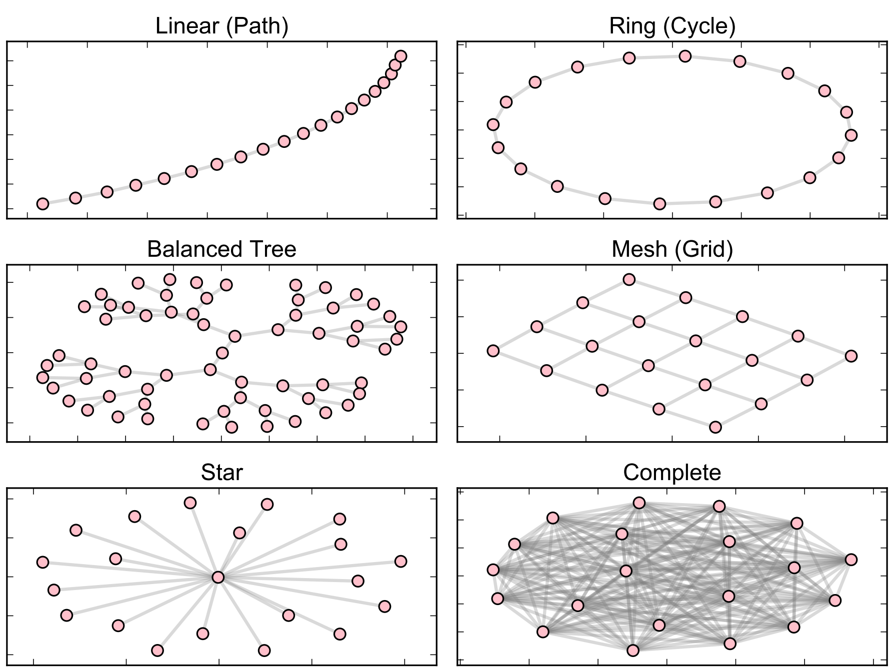
10. snap.stanford.edu/data/email-Enron.html

The second issue is that communication networks are often less rigorously documented than other types of social networks, or it takes more effort to access documentation. Verbal communications are almost never recorded. Telephone billing records may be available if requested by law enforcement officers, but not by social network analysts. Internet chat sessions are compulsorily recorded in some countries (such as Russia) but, again, not by "us" but by "them." And the free-flying carrier pigeons are provably the worst to track.

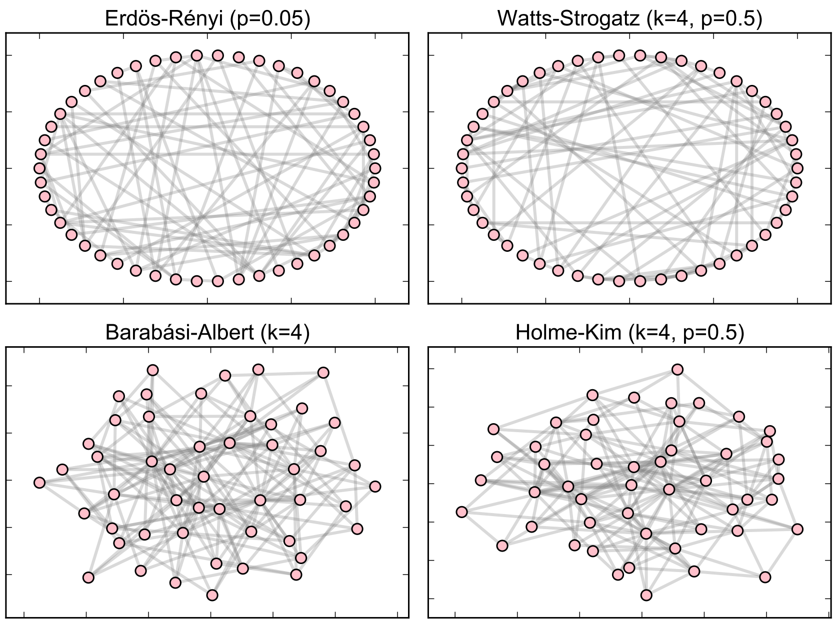
Appreciate Synthetic Networks

Synthetic networks are a cheap alternative to real-world, empiric networks. Unlike empiric networks that have to be scrupulously collected, either automatically or by hand, synthetic networks are generated by computer software (in our case, `NetworkX`). With proper adjustment through the right choice of the parameters of synthetic graph generators, you can produce networks of almost any type, resembling empiric networks to the point of confusion.

The following figure shows six generated "classic" networks. You saw almost all of them (except for the complete graph) in *Know Thy Networks*, on page 2. However, that figure was hand-programmed, but the following one is produced by the `NetworkX` graph generator functions. You will learn how to use them in *Generate Synthetic Networks*, on page 78.



It is worth remembering that these six networks are *not* complex because they have a predictable, regular, and easily describable structure. But the networks in the following figure are random, though defined by four different random models. (The models are properly described in *Network Analysis: Methodological Foundations [BE05]*.)



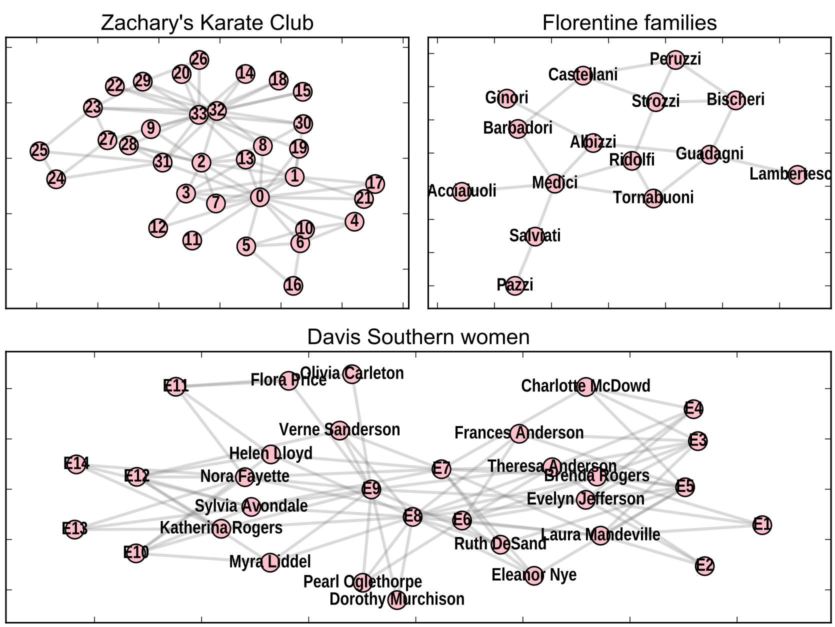
An undirected Erdös–Rényi graph, also known as a binomial graph, contains `N` nodes. It could have up to `N×(N-1)/2` edges, but each edge is instantiated with the probability of `p`. As a result, the expected number of edges is `p×N×(N-1)/2`. If `p==0`, then the network falls apart into isolated nodes. If `p==1`, the network becomes a complete graph. Note that in general, a node is not connected to its geometric neighbors.

If you have no definite idea about what kind of network you want, use the Erdös–Rényi model. However, a Watts–Strogatz graph is a much more realistic approximation of a real-world social network. The model arranges `N` nodes in a ring, connects each node to `k` ring neighbors, and then "rewires" any edge —reconnects one of its ends to a randomly chosen node—with the probability of `p`. The rewired edges typically go across the ring. They create an illusion of a "small world," where geometrically remote nodes may be connected with a short path. The model explains the phenomenon of "six degrees of separation," which claims that, on average, any two people on Earth are only *six handshakes apart from each other [Wat03]*.

Unfortunately, no matter how you twist them, nodes in Watts–Strogatz networks do not form tight communities, and this makes "small-world" networks somewhat unrealistic, too. The *Barabási–Albert preferential attachment model [Bar03]* adds another level of realism to synthetic social networks. It uses the principle of preferential attachment mentioned briefly on page 5 and more in detail in *What Makes Components Giant?*, on page 129. When a new node is about to join an existing network, it is likely to make `k` connections to the nodes with the highest degree. The model stimulates the emergence of hubs —"celebrity" nodes with disproportionately many connections.

The Holme–Kim model goes one step further. After adding `k` edges, it also adds triads (introduced on page 6) with the probability of `p`, making the synthetic network even more clustered and lifelike.

Over the long history of social network analysis, several empiric networks were so frequently used in case studies that they became the "gold standard" of a small social network, very much like the "Hello, world!" programs in computer programming. For any practical purpose, they are almost as good as synthetic networks, except that they are really tiny. (But you cannot blame the researchers who constructed them! It was the time before `NetworkX` and even before personal computers.) The following figure shows three famous social networks: Zachary's Karate Club, Davis Southern women and the events they attend, and marriages in Florentine families.



Whether you assembled a social network by hand, by running a program, or by downloading a publicly available network graph from SNAP or another depository, remember: no two friendships, no two kinships, and no two human interactions are the same. That is to say, no two links in a social network are the same, but some are weak, and some are strong. Why does it matter?

Distinguish Strong and Weak Ties

Hardcore social network researchers call social network graph edges "ties." In this section, I will sometimes refer to edges as ties—mostly out of respect to Mark Granovetter who *introduced [Gra73]* and *elaborated [Gra83]* the concept of weak and strong ties and later showed that weak ties are "strong." In a sense, he was the first social researcher to consider weighted social networks.

Granovetter did not propose a mechanism for quantifying the strength of a tie, but offered four criteria to consider:

- The amount of time spent in the tie
- Emotional intensity
- Intimacy (mutual trust)
- Reciprocity

You may be able to evaluate the first and the last criteria, but the middle two require looking into the contents of the messages, which cannot be done if only message signatures are available. Nonetheless, combined with sentiment analysis and other natural language processing techniques (see, for example, *Natural Language Processing with Python [BKL09]*), it may be possible to develop a formal mechanism for assessing tie strength.

Granovetter argues that weak and strong ties have different vocations in social networks. Strong ties (such as those between spouses and close friends) tend to draw nodes together into tight, densely interconnected clusters—cliques (*Extract Cliques, on page 131*). Cliques often act as "knowledge reservoirs": anything said by any actor in a clique is overheard and presumably remembered by all other clique members. If any clique member forgets anything, the clique as a whole can easily reconstruct the missing knowledge.

Cliques are great at knowledge preservation but not good at knowledge generation. Since different cliques preserve different types of knowledge, connections between the "reservoirs" enable sharing. Such links are called bridges. You will learn how to detect bridges in *Betweenness Centrality, on page 93*. Not surprisingly, weak ties often serve as bridges, and that's why they are "strong." And, regardless of their function in a social network, from complex

network analysis' perspective, both weak and strong ties are just weighted edges with appropriately selected weights.

Social networks are a special breed of complex networks, limited to individuals (humans and animals) and organizations. They have been extensively studied in social and behavioral sciences. Social network analysis served as a precursor to complex network analysis. Complex networks often have thousands and millions of nodes, and depend on edges having different weights. In the next chapter, you will unlock `NetworkX` tools for the efficient construction of massive networks, including synthetic and weighted networks.

Here you see we have a very advanced form of drawing, and a form I should not advise you to employ in your early efforts to do professional work.

> *Ernest Knaufft, American editor and director of the Chautauqua Society of Fine Arts*

CHAPTER 7

Mastering Advanced Network Construction

Complex networks are rarely constructed one node and one edge at a time. Instead, they are generated from matrix data, edge lists, node dictionaries, probability distributions, and other native Python and third-party data structures. As a complex network analyst, you need to be familiar with the NetworkX interfaces to the real world.

In this chapter, you will learn how to convert Python and third-party data structures (namely, Pandas DataFrames and Pandas NumPy matrices) into NetworkX graphs and back, and how to generate synthetic networks.

Create Networks from Adjacency and Incidence Matrices

This section uses Pandas, NumPy.

Mathematical graphs as collections of nodes and edges are not the only way to represent complex networks. Researchers and practitioners often use tabular (matrix) data to describe networks. The two most popular matrix-based descriptions are adjacency and incidence matrices. (You may want to remind yourself of the definitions of adjacency and incidence on the bulleted list on page 17.)

Adjacency Matrix, the Python Way

An adjacency matrix A is a square N×N matrix, where N is the size of the graph to be defined. The row and column indexes indicate the source and target nodes, respectively. Depending on the network type, the acceptable range, properties, and interpretation of the matrix elements differ. If a network belongs to more than one type (say, weighted and directed), consider all relevant properties and interpretations (see table on page 70).

Network type	Adjacency matrix	Interpretation
Simple	Only 0s or 1s, and no 1s on the main diagonal	Absence/presence of an edge
Weighted	At least one floating-point number	Numbers are edge weights.
With self-loops	At least one non-zero on the main diagonal	Same as above
Signed	At least one negative number	Same as above
Undirected	Symmetric	Same as above
Multigraph	Not possible	Cannot be represented as an adjacency matrix

As an example, here's an adjacency matrix for the linear timeline of Abraham Lincoln from the figure on page 4 (left) and another very similar network (right):

```
0 1 0 0 0   0 1 0 0 0
0 0 1 0 0   0 0 1 0 0
0 0 0 1 0   0 0 0 1 0
0 0 0 0 1   0 0 0 0 1
0 0 0 0 0   1 0 0 0 0
```

The networks have five nodes each (the matrices are 5×5). The left network has four edges (the matrix has four 1s), and the right network has an extra edge. The networks are simple (the matrices have only 0s and 1s, and no 1s are on the main diagonal), unweighted, and unsigned. The networks are directed (the matrices are not symmetric). The additional 1 in the lower-left corner of the matrix on the right converts the linear network into a ring by connecting the last event (death) to the first event (birth). If Abe Lincoln had believed in reincarnations, he would have chosen the right matrix.

As a side note, the sum of all 1s in any column or row of an adjacency matrix equals the indegree or outdegree, respectively, of the corresponding node.

The most common way of representing matrices in pure Python is in the form of a list of lists. The right previous matrix is a list of five lists, one list per row. Suppose it is given to us (say, produced by another function elsewhere in our program):

```
A = [[0,  1, 0, 0, 0],
     [0,  0, 1, 0, 0],
     [0,  0, 0, 1, 0],
     [0,  0, 0, 0, 1],
     [.1, 0, 0, 0, 0]]
```

Note that since the reincarnation is not inevitable at all, not even in the case of Honest Abe, we set the weight of the death-to-birth edge to 0.1.

How can we convert this matrix to a graph? There are at least three ways: one uses pure Python and NetworkX, and the other two rely on NumPy (*When Python Goes Numerical*, on page 71) and Pandas (*Another Kind of Pandas*, on page 73).

When Python Goes Numerical

Surely, Python, just like all other computer software, internally works with numbers, only numbers, and nothing but numbers. However, when there are too many numbers to work with, Python performance significantly degrades. NumPy ("Numerical Python") is a package for scientific computing that accelerates fundamental numerical operations. It provides support for multidimensional objects (such as vectors and matrices), vectorized arithmetic and algebraic operations, and other goodies. NumPy is a part of the SciPy ("Scientific Python") ecosystem that also includes Pandas for data science, Matplotlib for plotting, Sympy for symbolic computations, and IPython for interactive development.

If performance is not an issue (if your network has fewer than a couple of thousand nodes), the pure Python solution may make you feel more comfortable, especially if you have never used NumPy. Remember that any non-zero element in the adjacency matrix represents an edge from the "row node" to the "column node." Create an empty directed graph, enumerate each matrix element twice (by rows and then by columns), and extract non-zero elements. Their indexes represent network edges, which you can add to the graph by calling G.add_edges_from().

```
from itertools import chain # For flattening the list of edges
G = nx.DiGraph()
edges = chain.from_iterable([(i, j)
                             for j, column in enumerate(row)
                             if A[i][j]] for i, row in enumerate(A))
G.add_edges_from(edges)
print(G.edges(data=True))
```

```
[(0, 1, {}), (1, 2, {}), (2, 3, {}), (3, 4, {}), (4, 0, {})]
```

By default, NetworkX assumes that all edges have the weight of 1, and does not display weights as edge attributes. If the matrix represents signed or unsigned weights (rather than absence/presence), you can modify the code to incorporate the "weight" attribute:

```
from itertools import chain # For flattening the list of edges
G = nx.DiGraph()
edges = chain.from_iterable([(i, j, {"weight": A[i][j]})
                             for j, column in enumerate(row)
                             if A[i][j]] for i, row in enumerate(A))
G.add_edges_from(edges)
print(G.edges(data=True))
```

```
❮ [(0, 1, {'weight': 1}), (1, 2, {'weight': 1}), (2, 3, {'weight': 1}),
   (3, 4, {'weight': 1}), (4, 0, {'weight': 0.1})]
```

Adjacency Matrix, the NumPy Way

The NumPy way is somewhat more concise, but you must convert the list of lists to a 2D matrix and give NetworkX a hint about the network type.

```
import numpy as np
A_mtx = np.matrix(A)
G = nx.from_numpy_matrix(A_mtx, create_using=nx.DiGraph())
print(G.edges(data=True))
```

```
❮ [(0, 1, {'weight': 1}), (1, 2, {'weight': 1}), (2, 3, {'weight': 1}),
   (3, 4, {'weight': 1}), (4, 0, {'weight': 0.1})]
```

As a bonus, the NumPy way is significantly faster for large networks. Also, note how NumPy intelligently treated matrix elements as edge weights!

You can program the reverse transformation with nx.to_numpy_matrix(G):

```
B_mtx = nx.to_numpy_matrix(G) # Produces a NumPy 2D matrix
print(B_mtx)
```

```
❮ [[ 0.   1.   0.   0.   0.]
   [ 0.   0.   1.   0.   0.]
   [ 0.   0.   0.   1.   0.]
   [ 0.   0.   0.   0.   1.]
   [ 0.1  0.   0.   0.   0.]]
```

To convert the matrix back to a list of lists, call method tolist():

```
B_lst = B_mtx.tolist()
print(B_lst)
```

```
❮ [[0.0, 1.0, 0.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0, 0.0],
   [0.0, 0.0, 0.0, 1.0, 0.0], [0.0, 0.0, 0.0, 0.0, 1.0],
   [0.1, 0.0, 0.0, 0.0, 0.0]]
```

By the way, you may have just learned how to program with NumPy.

Adjacency Matrix, the Pandas Way

The most versatile connection to date is between NetworkX and Pandas. If you consider integrating CNA with general data scientific methods—regression, prediction, and classification—you must be aware of the interface between NetworkX and Pandas.

Another Kind of Pandas

Pandas are cute. Pandas is also yet another component of the SciPy ("Scientific Python") ecosystem. Its main application is data science, and it provides a basketful of data structures and algorithms for storing and processing labeled rectangular data. The most famous Pandas data structures are a Series (a labeled vector) and a DataFrame (a labeled table). You can read more about Pandas and NumPy in *Data Science Essentials in Python [Zin16]*.

Converting a NetworkX graph to a Pandas adjacency matrix costs one function call, just like almost any other popular operation in Pandas. Before we do so, let's first relabel the graph nodes to allow at least some meaningful interpretation:

```
labels = "Born", "Married", "Elected Rep", "Elected Pres", "Died"
nx.relabel_nodes(G, dict(enumerate(labels)), copy=False)
df = nx.to_pandas_dataframe(G)
print(df)
print(type(df))
```

```
              Died  Elected Rep  Married  Born  Elected Pres
Died           0.0          0.0      0.0   0.1           0.0
Elected Rep    0.0          0.0      0.0   0.0           1.0
Married        0.0          1.0      0.0   0.0           0.0
Born           0.0          0.0      1.0   0.0           0.0
Elected Pres   1.0          0.0      0.0   0.0           0.0
<class 'pandas.core.frame.DataFrame'>
```

Discussing the uses of DataFrame objects is beyond the scope of this book; try this adventure on your own!

Suddenly, function nx.from_pandas_dataframe(df,source,target) is not a counterpart of nx.to_pandas_dataframe() in the same sense as nx.from_numpy_matrix(df) is a counterpart of nx.to_numpy_matrix(). The first parameter of nx.from_numpy_matrix() is a DataFrame that represents the adjacency matrix. The first parameter of nx.from_pandas_dataframe() is a DataFrame, too, but each row defines one edge, and two of the columns, source and target, designate the start and end nodes of the edge. (You can convert the remaining columns into edge attributes.) This

approach is more flexible than the adjacency matrix. For example, it easily allows parallel edges.

Let's build Honest Abe's lifetime network from a data frame. First, create a data frame—in the real world, it would be an output of another part of the same program. Second, call nx.from_pandas_dataframe(). No magic involved.

```
import pandas as pd
df = pd.DataFrame({
  "from": {0: "Died", 1: "Elected Rep", 2: "Married", 3: "Born",
           4: "Elected Pres"},
  "to": {0: "Born", 1: "Elected Pres", 2: "Elected Rep", 3: "Married",
         4: "Died"},
  "weight": {0: 0.1, 1: 1.0, 2: 1.0, 3: 1.0, 4: 1.0},
  })
print(df)
```

```
           from            to  weight
0          Died          Born     0.1
1   Elected Rep  Elected Pres     1.0
2       Married   Elected Rep     1.0
3          Born       Married     1.0
4  Elected Pres          Died     1.0
```

```
G = nx.from_pandas_dataframe(df, "from", "to", edge_attr=["weight"])
print(G.edges(data=True))
```

```
[('Born', 'Married', {'weight': 1.0}), ('Born', 'Died', {'weight': 0.1}),
 ('Married', 'Elected Rep', {'weight': 1.0}),
 ('Elected Pres', 'Died', {'weight': 1.0}),
 ('Elected Pres', 'Elected Rep', {'weight': 1.0})]
```

By the way, you may have just learned how to program with Pandas.

Handling Node Attributes, the Pandas Way

Another case of collaboration between Pandas and NetworkX is importing node attributes into a DataFrame. In the course of CNA, you often decorate network nodes with various attributes: labels, weights, centralities, demographics (age, gender), and the like. For the sake of experimentation, let's add a "date" parameter to Lincoln's timeline:

```
events = {"Died": 1865, "Born": 1809, "Elected Rep": 1847,
          "Elected Pres": 1861, "Married": 1842}
nx.set_node_attributes(G, "date", events)
node_data = G.nodes(data=True)
print(node_data)
```

```
[('Died', {'date': 1865}), ('Elected Rep', {'date': 1847}),
 ('Married', {'date': 1842}), ('Born', {'date': 1809}),
 ('Elected Pres', {'date': 1861})]
```

The values of node attributes, especially those calculated by the analysis program, may become inputs to further data analysis steps, as mentioned in *Adjacency Matrix, the Pandas Way*, on page 73. If you're a Pandas person, you should move the node attributes from NetworkX to a DataFrame. Luckily, one of the DataFrame constructors builds a DataFrame from a list of tuples, and node_data is a list of tuples.

```
lincoln_ser = pd.DataFrame(node_data).set_index(0)[1]
print(lincoln_ser)
```

```
0
Died            {'date': 1865}
Elected Rep     {'date': 1847}
Married         {'date': 1842}
Born            {'date': 1809}
Elected Pres    {'date': 1861}
Name: 1, dtype: object
```

After converting the node labels to the row index, the resulting DataFrame has only one column named 1 (which, naturally, is a Series). The values in the column are node attribute dictionaries, and one of the Series constructors builds a Series from a dictionary. Let's apply the constructor to each row.

```
df = lincoln_ser.apply(pd.Series)
print(df)
```

```
              date
0
Died          1865
Elected Rep   1847
Married       1842
Born          1809
Elected Pres  1861
```

The result is a DataFrame suitable for further processing. For example, you can calculate the duration, in years, of each span of Lincoln's biography:

```
spans = df.sort_values('date').diff()
print(spand)
```

```
              date
0
Born           NaN
Married       33.0
Elected Rep    5.0
Elected Pres  14.0
Died           4.0
```

(NaN, "not a number," is a Pandas way of reporting a missing or otherwise unavailable value.)

Incidence Matrix

An incidence matrix J is a rectangular N×M matrix, where N is the number of nodes and M is the number of edges. A 1 at J[i,j] means that the node i is incident to the edge j. All other elements of J are 0s. If the represented graph is directed, the start node is designated with 1 and the end node with -1.

Unlike an adjacency matrix, an incidence matrix easily allows parallel edges. However, it has its weak points: weighted networks cannot be represented, and an incidence matrix of a typical complex network has a larger memory footprint than the adjacency matrix of the same network.

Function nx.incidence_matrix(G) returns the incidence matrix of G as a so-called sparse matrix. (Pass the optional parameter oriented=True to distinguish start and end nodes.) You can convert a sparse matrix to a dense one with G.todense():

```
J = nx.incidence_matrix(G, oriented=True).todense()
print(J)
```

```
[[-1.  0.  0.  0.  1.]
 [ 1. -1.  0.  0.  0.]
 [ 0.  1. -1.  0.  0.]
 [ 0.  0.  1. -1.  0.]
 [ 0.  0.  0.  1. -1.]]
```

Here's how we read the results: edge number 0 starts at node 1 (because J[1,0]==1) and ends at node 0 (because J[0,0]==-1); edge number 1 starts at node 2 (because J[2,1]==1) and ends at node 1 (because J[1,1]==-1), and so on.

Work with Edge Lists and Node Dictionaries

You do not have to mess with matrices, NumPy, and Pandas to bulk move data between your code and NetworkX networks. You can use edge lists and node dictionaries.

Edge Lists

An edge list is a list of 3-tuples containing the start node, end node, and a dictionary of edge attributes for each edge. You can obtain it from an existing network by calling nx.to_edgelist() or construct it yourself and feed as the parameter to nx.from_edgelist() to produce a new network.

```
edges = nx.to_edgelist(G)
F = nx.from_edgelist(edges, create_using=G)
print(F.edges(data=True))
```

```
❮ [('Born', 'Married', {'weight': 1}), ('Married', 'Elected Rep',
   {'weight': 1}), ('Elected Pres', 'Died', {'weight': 1}),
   ('Died', 'Born', {'weight': 0.1}), ('Elected Rep', 'Elected Pres',
   {'weight': 1})]
```

Incidentally (or not), the value returned by G.edges(data=True) (on page 21) is equivalent to the value returned by nx.to_edgelist(). That is to say, one of the two functions are redundant.

The pair of the edge list-related functions is reversible: a graph A, created from an edge list extracted from another graph B, is equal to B. Equality of graphs in mathematical graph theory is called isomorphism. Two graphs are isomorphic if you align all of the nodes of one graph with all of the nodes of the other graph, and all of their edges will align, too. This property is good enough for the graphs with unlabeled nodes but too weak for real-world labeled graphs. Yet, for the lack of a better tool, let's use the function nx.is_isomorphic():

```
print(nx.is_isomorphic(F, G))
```

```
❮ True
```

Dictionary of Lists

A dictionary of lists of nodes is what it says it is. All nodes in a graph are the keys, and lists of adjacent nodes are values. You can get a dictionary of lists with nx.to_dict_of_lists():

```
dict_list = nx.to_dict_of_lists(G)
print(dict_list)
```

```
❮ {'Born': ['Married'], 'Married': ['Elected Rep'], 'Elected Pres': ['Died'],
   'Died': ['Born'], 'Elected Rep': ['Elected Pres']}
```

nx.to_dict_of_lists() does not externalize edge attributes, including width, and this makes the resulting dictionary unsuitable for recreating the original graph with nx.from_dict_of_lists(). It is true that the new graph is isomorphic to the source, but the function nx.is_isomorphic() looks only at the topology of the graphs and does not compare the attributes.

```
F = nx.from_dict_of_lists(dict_list, create_using=G)
nx.is_isomorphic(F, G)
```

```
❮ True
```

The dictionary-of-lists mechanism does not appear to be well thought out. You may be better off with accessing the network edge dictionary (which is a dictionary of dictionaries) directly:

```
print(G.edge)
```

```
{'Born': {'Married': {'weight': 1}}, 'Married': {'Elected Rep':
 {'weight': 1}}, 'Elected Pres': {'Died': {'weight': 1}},
 'Died': {'Born': {'weight': 0.1}}, 'Elected Rep': {'Elected Pres':
 {'weight': 1}}}
```

Generate Synthetic Networks

You have read in *Appreciate Synthetic Networks*, on page 63, that not only can networks be built from experimental, real-world data, but they can also be synthesized. Synthetic networks can be regular (constructed by executing deterministic algorithms) or complex (emerge from probability distributions). NetworkX functions that build synthetic network graphs are called graph generators (not to be confused with Python generator objects).

In 1999, Ronald C. Read and Robin J. Wilson published a *collection of 10,000 small synthetic graphs [RW99]*. NetworkX provides generators for 1,253 of them —and about 110 more regular ("classic") and complex networks. At the moment, it suffices to look only at the graph generators whose output is shown in the figures on page 63, on page 64, and on page 65. Let's start with the "classic" networks: paths, cycles, stars, complete graphs, trees, and grids.

generators.py

```
# Generate and draw classic networks
G0 = nx.    path_graph(20)
G1 = nx.   cycle_graph(20)
G4 = nx.    star_graph(20)
G5 = nx.complete_graph(20)
G2 = nx. balanced_tree(2, 5)
G3 = nx. grid_2d_graph(5, 4)
names = ("Linear (Path)", "Ring (Cycle)", "Balanced Tree", "Mesh (Grid)",
         "Star", "Complete")
```

The first four functions need to know the total number of nodes. There is only one way to generate the edges for these types of graphs. For a balanced tree, you must provide the branching factor r (the number of children of a non-leaf node) and the height h (the height does not include the root node of the tree). A balanced tree has r^{h+1}-1 nodes. In our example, G2 is a five-level binary tree with 2^{5+1}-1=63 nodes. To build a two-dimensional grid (mesh) like G3, specify the number of rows n and columns m, and get a graph with m×n nodes.

The next example shows the use of generators for Erdös–Rényi (really random), Watts–Strogatz (small world), Barabási–Albert (preferential attachment), and Holme–Kim (enhanced preferential attachment) random graphs.

generators.py

```
# Generate and draw random networks
G0 = nx.             erdos_renyi_graph(50,    0.05)
G1 = nx.connected_watts_strogatz_graph(50, 4, 0.5 )
G2 = nx.         barabasi_albert_graph(50, 4      )
G3 = nx.        powerlaw_cluster_graph(50, 4, 0.5 )
names = ("Erdös-Rényi (p=0.05)", "Watts-Strogatz (k=4, p=0.5)",
         "Barabási-Albert (k=4)", "Holme-Kim (k=4, p=0.5)")
```

All four functions need to know the total graph size. The remaining parameters characterize the random nature of the interconnecting edges:

- For *Erdös–Rényi*: the probability of edge creation. Incidentally, it equals the graph density (*Start with Global Measures*, on page 83).
- For *Watts–Strogatz*: the initial number of neighbors and the probability of edge rewiring
- For *Barabási–Albert*: the number of edges to attach from a new node
- For *Holme–Kim*: the same as above, plus the probability of adding a triangle for each added edge

The remaining three generators produce "famous" social networks that were initially constructed by field sociologists based on experimental data, but eventually became "gold standards" of social network research.

generators.py

```
# Generate and draw famous social networks
G0 = nx.karate_club_graph()
G1 = nx.davis_southern_women_graph()
G2 = nx.florentine_families_graph()
names = ("Zachary's Karate Club", "Davis Southern women",
         "Florentine families")
```

The networks, though formerly random, are stored by NetworkX in the form of fixed edge lists. Their generators need no parameters.

Slice Weighted Networks

Lucky network analysts work with unweighted networks. In an unweighted network, all edges are equal. You consider either all of them, and get what you get—or none of them, and get a network with no edges.

Unlucky network analysts work with weighted (and possibly signed) networks. In a weighted network, some edges are strong, and some are weak. If you keep all edges, you will have a distorted view of the network because there are algorithms that do not discriminate edges by weight. For them, an edge with a weight of 1.00 (to your best life-long friend) has the same importance

as another edge with a weight of 0.01 (to the guy who takes the same 7:00 a.m. bus, always sits in the back, and reads *Alaska Dispatch News*).

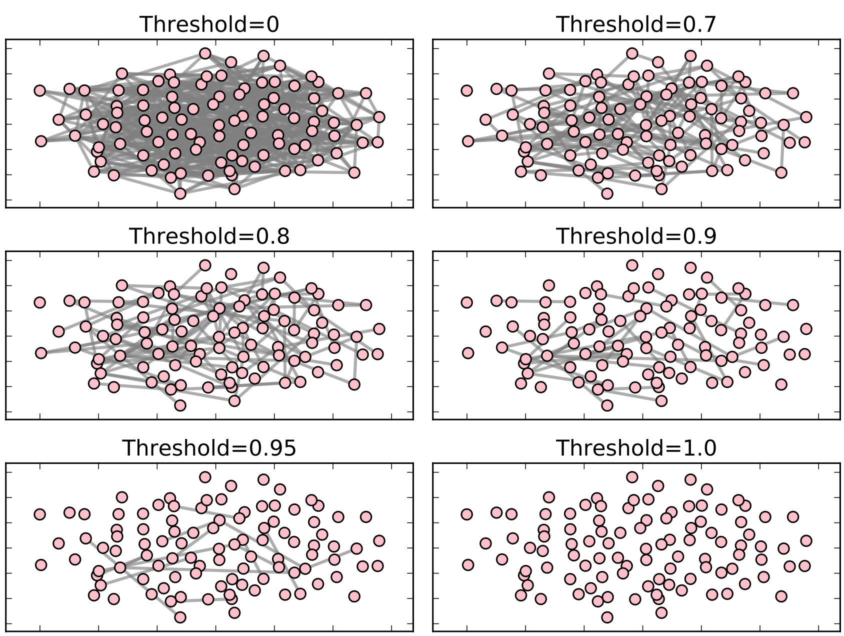
Most network analysts are unlucky and have to slice their networks.

Slicing is the process of eliminating low-strength edges (weak ties). In the simplest form, you choose a cut-off threshold `T` that controls the density of the resulting network. Each edge's weight is compared to the threshold. If the weight is at or above the threshold, the edge remains in the network; otherwise, it is erased.

`NetworkX` does not provide a standard slicing routine, but you can quickly implement yours (will do later). However, first, you should decide on the value of `T`. If the cut-off is too high, the network falls apart into tiny disjoint fragments; if it is too low, the network becomes a hairball with no analyzable structure. The trial-and-error approach may be the best:

1. Select a `T` based, say, on the edge weight distribution.
2. Slice the network.
3. Get some measurements (the number of fragments, density, and so on).
4. If the results do not suit you, go back to square one.

The following figure shows the same Erdös–Rényi random network with one hundred nodes, sliced with six different thresholds.



The final value of T depends on the network topology and weight distribution.

Our new function, slice_network(G, T, copy=True), retrieves all weighted edges from the network G, identifies "underweighted" edges, and removes them. The function by default operates on the copy of G, allowing us to step back and give another try without rebuilding G.

slicing.py

```
def slice_network(G, T, copy=True):
    """
    Remove all edges with weight<T from G or its copy.
    """
    F = G.copy() if copy else G
    F.remove_edges_from((n1, n2) for n1, n2, w in F.edges(data="weight")
                        if w < T)
    return F

F = slice_network(G, 0.9)
print(F.edges())
```

```
[('Elected Rep', 'Elected Pres'), ('Married', 'Elected Rep'),
 ('Born', 'Married'), ('Elected Pres', 'Died')]
```

The possibility of reincarnation is no more.

Now you know how to create a complex network of any size from any dataset and how to convert the network structure to the most popular pure Python and third-party data structures. We will next look into network measuring tools.

Considering the value for clearness of thought of counting, measuring and weighing, it is not surprising to find that in the seventeenth century, and even at the end of the sixteenth, the advance of the sciences was accompanied by increased exactness of measurement and by the invention of instruments of precision.

➤ *Walter Libby, American writer*

CHAPTER 8

Measuring Networks

Almost everything you have seen so far in this book has been about constructing complex networks, not about analyzing them. In other words, it was CN, but not CNA. This chapter delves into CNA and introduces some important CNA toolsets. You will learn how to measure dyadic, triadic, and global properties of network nodes: distances, loops, clustering coefficient, assortativity, and a variety of centralities. You will be able to identify the most central nodes and interpret their importance. You will be able to locate network regions that differ in local density and attribute uniformity.

Start with Global Measures

Let's start with a "black box" view of a complex network. Let's pretend we are at a distance and instead of nodes, edges, and their attributes, we see a fuzzy grayish cloud. What can we tell about that cloud? Not much: only its size and density.

To be specific, in this chapter, we will experiment with the network of CNA-related Wikipedia pages constructed in Chapter 5, *Case Study: Constructing a Network of Wikipedia Pages*, on page 41 and available in the file cna.graphml.[1]

The size of a network is either its node count or edge count. You can measure both using the standard Python len() function and other specialized functions.

```
len(G) # Number of nodes
len(G.node)
len(G.nodes())
nx.number_of_nodes(G)
len(G.edge())
```

```
❮ 2988
```

1. pragprog.com/titles/dzcnapy/source_code

```
len(G.edges()) # Number of edges
nx.number_of_edges(G)
```

```
❮ 11545
```

Note that `len(G.edge())` returns the number of *nodes*, not *edges*, because `G.edge()` returns a dictionary of neighbors with one entry per node.

If you're curious, you can also get the number of non-existent edges—the edges that could connect two nodes but don't. The function `nx.non_edges()` returns a Python generator of missing edges. Before measuring it, you must convert it to a list. Beware: most real-life graphs have orders of magnitude more missing edges than present edges. Your computer may quickly run out of memory if you attempt to make a list out of them.

```
len(list(nx.non_edges(G)))
```

```
❮ 8913611
```

Graph density measures the fraction of existing edges out of all potentially possible edges. Density is a number between 0 and 1, inclusive. A network with density 0 has no edges whatsoever. A network with density 1 is a complete graph. For a directed network with `n` nodes and `m` edges, density is calculated as `m/(n(n-1))`; for undirected networks, it is calculated as `2m/(n(n-1))`, because, compared to directed networks, they have only half of potentially possible edges. You can measure density by calling a namesake function.

```
nx.density(G)
```

```
❮ 0.0012935348132850563
```

The density of the Wikipedia network is low—only about 0.1 percent. Only one out of about 1,000 possible edges exists in the graph. This value is not unusual: most complex networks have similarly low density.

Explore Neighborhoods

Node and edge counts and density are some of the macroscopic network properties. Let us now zoom into a network and look at it at the microscopic level—at the level of individual nodes and their neighbors.

The network neighborhood of a node is the set of all nodes adjacent to that node. Social network analysis pays particular attention to neighborhoods because that is where we find the relatives, close friends, and colleagues of the actor represented by the central node—in other words, the most socially significant *alters* of the *ego*. (Check *Egocentric Networks*, on page 54, to refresh

the meaning of the emphasized words.) Neighborhoods are responsible for the local properties of network graphs.

NetworkX offers two mechanisms for calculating neighborhoods. (To be specific, let's compute the neighborhood of the node ego="Neighbourhood (Graph Theory)".)

- Use the implicit dictionary representation of the graph. Node names are keys, and adjacent node dictionaries are values.

```
alters1 = G[ego]
print(alters1)
print(len(alters1))
```

```
❮ {'Turán Graph': {}, 'Isolated Vertex': {}, 'Adjacency List': {},
   'Graph (Discrete Mathematics)': {}, 'Complement Graph': {},
   'Journal Of The Acm': {}, 'Triangle Free Graph': {}, 'Dense Graph': {},
   'Vertex (Graph Theory)': {}, 'Loop (Graph Theory)': {},
   'Linear Time': {}, 'Planar Graph': {}, 'Vertex Figure': {},
  «...»
   'Independent Set (Graph Theory)': {}, 'Claw Free Graph': {},
   'Discrete Mathematics (Journal)': {}, 'Cycle Graph': {}}
  35
```

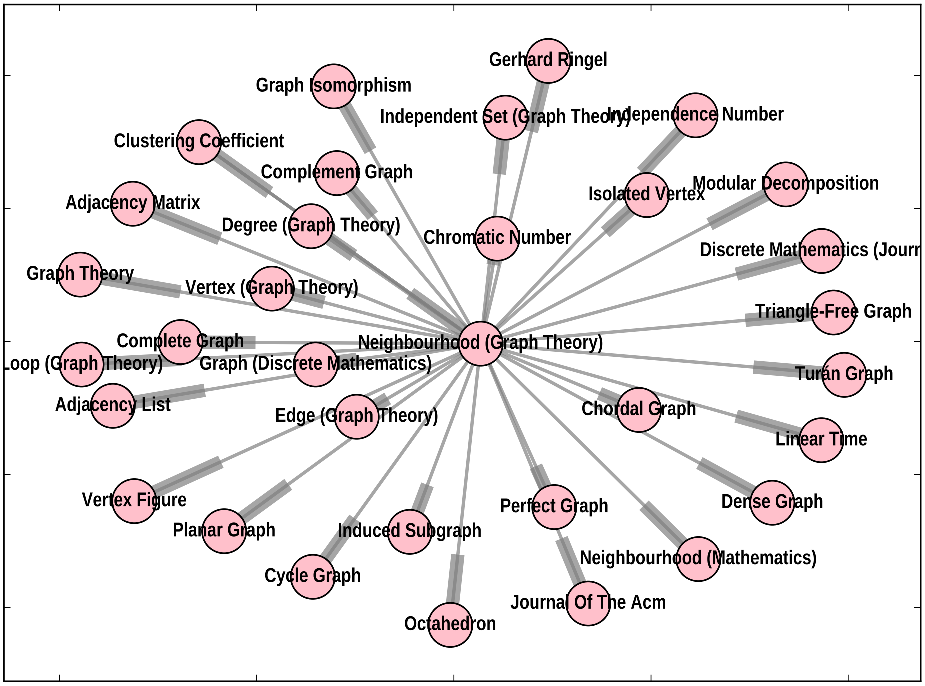
  The empty dictionaries {} would hold edge attributes if the network had edge attributes.

- Call the function nx.all_neighbors(). The function returns a generator object that you can convert to a list. However, if you expect a node to have too many neighbors and you do need all of them at once, keep the neighborhood in the generator form until later.

```
alters2 = list(nx.all_neighbors(G, ego))
print(alters2)
print(len(alters2))
```

```
❮ ['Watts And Strogatz Model', 'Network Science', 'Spatial Network',
   'Scientific Collaboration Network', 'Semantic Network',
   'Barabási–Albert Model', 'Reciprocity (Network Science)',
   'Biological Network', 'Clustering Coefficient', 'Pavol Hell',
  «...»
   'Graph Isomorphism', 'Modular Decomposition', 'Planar Graph',
   'Vertex Figure', 'Independent Set (Graph Theory)', 'Cycle Graph',
   'Discrete Mathematics (Journal)', 'Claw Free Graph']
  65
```

Neither neighborhood contains the ego node itself. Such neighborhoods are called "open." The figure on page 86 shows the out-neighborhood of ego: yet another star. Don't forget that the gray rectangles are Matplotlib's idea of arrows.

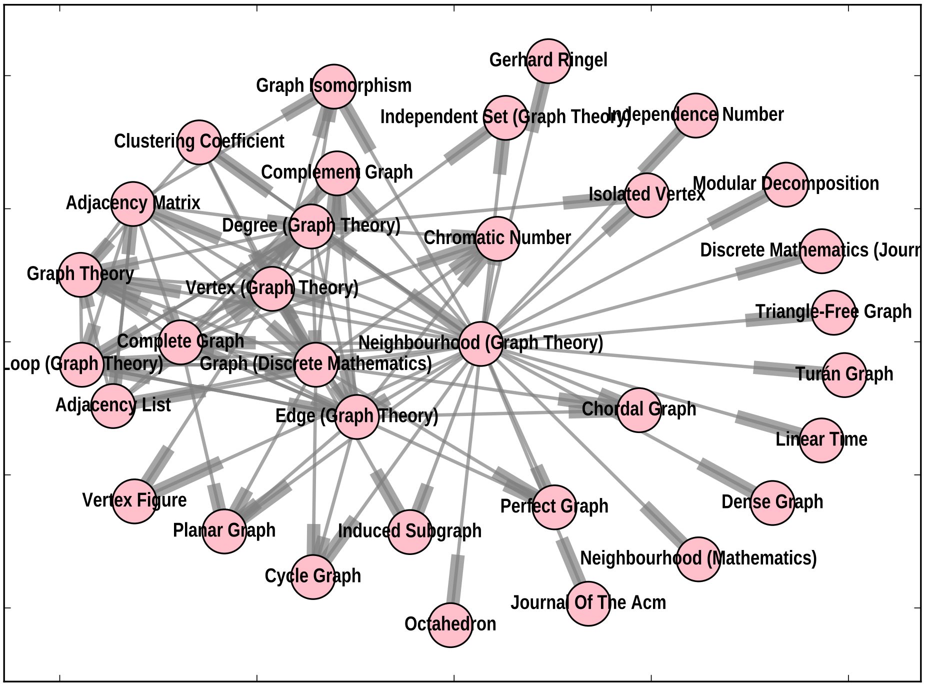


Note that the two methods report a different number of nodes in the neighborhoods `alters1` and `alters2`: 35 and 65, respectively. Recall that the network `G` is directed. The first method returns only the neighbors reachable by the outgoing edges—the out-neighborhood. The second method returns all adjacent nodes, regardless of the direction of adjacency. Which method to use depends on which result you're looking for.

A neighborhood is a dyadic structure. It's defined in terms of connections between two nodes: the *ego* and an *alter*. Aside from serving as a reference to the *ego*'s inner circle, it conveys little information. For example, it doesn't tell if and how its members are interconnected. Adding the chord edges transforms the sparse neighborhood into an egocentric network (*Egocentric Networks, on page 54*). Call function `nx.ego_graph()` to obtain the egocentric network graph.

```
egonet = nx.ego_graph(G, ego)
```

The figure on page 87 shows the egocentric network of `ego`. The network is much denser. You can see that some nodes are connected only to the hub (and possibly to some more remote nodes), while others form triangles that involve more neighborhood members.



Clustering Coefficient

Some social theories consider triads essential units of social network analysis. Function `nx.clustering(G, nodes=None)` calculates the clustering coefficient—a measure of the prevalence of triangles in an egocentric network. The clustering coefficient is the fraction of possible triangles that contain the `ego` node and exist. This measure is undefined for directed graphs; you must coerce a digraph to an undirected graph before calculating the clustering coefficient. The following code fragment shows how to call the function:

```
cc = nx.clustering(nx.Graph(G), ego)
print(cc)
```

```
0.36251920122887865
```

If the clustering coefficient of a node is 1, the node participates in every possible triangle involving any pair of its neighbors; the egocentric network of such a node is a complete graph. If the clustering coefficient of a node is 0, no two nodes in the neighborhood are connected; the egocentric network of such node is a star. Think of the clustering coefficient as a measure of "stardom."

Cluster Clusteri Lupus Est

Just like some other terms, the term "clustering" refers to at least three different concepts: the separation of a network into compact, tightly knit communities (*Outline Modularity-Based Communities*, on page 136); the measure of the density of an egocentric network; and the task of grouping relational data objects into subsets with similar properties.

Function nx.average_clustering() calculates the mean clustering coefficient for all nodes of a simple network (no loops, no directed or parallel edges).

```
acc = nx.average_clustering(nx.Graph(G))
print(acc)
```

```
❮ 0.7266398872539529
```

The average clustering coefficient is not to be confused with the clustering coefficient of the whole network—the fraction of all possible triangles that exist in the network. The latter is known as transitivity, a measure of transitive closure (explained on page 5). NetworkX has a namesake function to calculate it, too:

```
trans = nx.transitivity(G)
print(trans)
```

```
❮ 0.03412721874374035
```

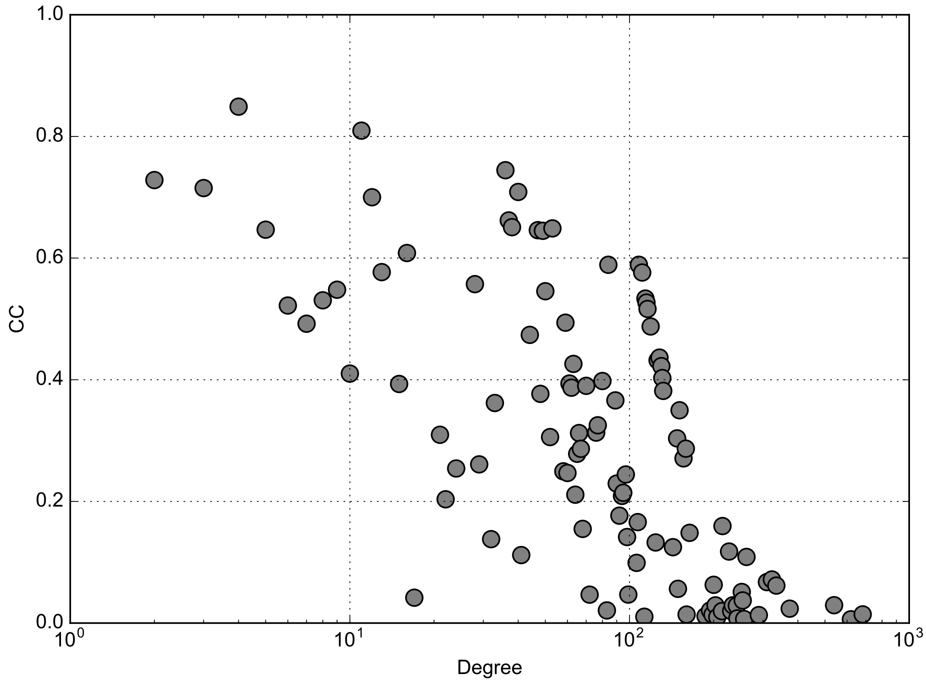
You can see the discrepancy between the two alternative measures of the "stardom." The source of the discrepancy is a considerable proportion of nodes with few neighbors. For such nodes, the local clustering coefficient is traditionally high, as shown in the figure on page 89, and it inflates the mean value.

By the way, the figure presents the results of a real, though so far concealed, exploratory complex network experiment. You will see the mechanics of similar experiments in *Choose the Right Centralities*, on page 92.

Think in Terms of Paths

Both dyadic and triadic relationships are local and never go farther than one edge (or one "hop," as network researchers say) from any of the involved nodes. The purpose of the functions in this section is to take you far away—as far as your network can afford.

For that, you need definitions of a walk, trail, and path.



- A walk in a network is any sequence of edges such that the end of one edge is always the beginning of another edge, except possibly for the first and last edges that may be connected only at one end.
- A trail is a walk that never uses the same edge twice. A trail that does not intersect itself, but starts and ends at the same node, is called a cycle (a self-loop edge explained on page 18 is a cycle).
- A path is a trail that never visits the same node twice (in other words, it does not intersect itself; `NetworkX` refers to paths as "simple paths").

Any of these walks is directed if any of its constituent edges is directed. For the rest of the book, we will use only paths.

A path has the length. The length of a path in an unweighted network is the number of edges in the path. When it comes to weighted paths, it is up to you to decide how to calculate the length. Possible metrics include the number of edges, the sum of the weights, the harmonic average of the weights, and the largest or the smallest weight.

A path is a highway across the network that indirectly connects the nodes that are not adjacent to each other, and allows them to interact. The meaning of the remote interaction is specific to each class of complex networks. In a social network, a path of length 3 connects the *ego* to the friend-of-a-friend-of-a-friend. In the network of Wikipedia pages, a path of length 3 connects the page about neighborhoods to a page that is similar to a similar page. In a transportation network, a node three hops away is the destination you can get to by changing trains twice. In some networks (like the network of foods and nutrients), long paths make no sense at all.

Two nodes in a network are often connected with more than one path. If paths matter at all, the shortest of them matters the most. (Again, there may be more than one shortest path between two nodes.) The shortest paths are called *geodesics*.

Not only does NetworkX provide a set of tools for computing with paths, but it also uses them for component detection and centrality calculation, to name a few applications. Function nx.shortest_simple_paths(G,u,v) returns a generator of all shortest paths between the nodes u and v. You can expand the generator into a list, but beware: it may take the program hours and even days to elicit all shortest paths in a large graph. Use this function with care! For example, you can get one path at a time by calling next().

```
path_gen = nx.shortest_simple_paths(G, ego, "Agent Based Model")
next(path_gen)
```

```
❮ ['Neighbourhood (Graph Theory)', 'Clustering Coefficient',
   'Social Network', 'Agent Based Model']
```

```
next(path_gen)
```

```
❮ ['Neighbourhood (Graph Theory)', 'Edge (Graph Theory)',
   'Small World Network', 'Agent Based Model']
```

```
next(path_gen)
```

```
❮ ['Neighbourhood (Graph Theory)', 'Vertex (Graph Theory)',
   'Semantic Network', 'Agent Based Model']
```

Function nx.shortest_path(G,source=None,target=None) returns only one of the shortest paths between source and target, but if you omit either or both of the parameters, it returns either all shortest paths starting at source, all shortest paths ending at target, or all shortest paths in the network.

```
path = nx.shortest_path(G, ego, "Agent Based Model")
```

```
❮ ['Neighbourhood (Graph Theory)', 'Clustering Coefficient',
   'Social Network', 'Agent Based Model']
```

Networks as Circles

Reportedly, people used to believe that the Earth was flat and round, but later changed their mind and settled on the geoid—an ellipsoid-like, but not exactly ellipsoidal body. We are going the opposite way: our complex networks started as shapeless clouds, but at this point let's try to treat them as flat and round.

CNA offers a concept of node eccentricity—a measure of how far from (or close to) the center a node is, wherever the center is. The eccentricity is the maximum distance from a node to all other nodes in the network. The distance between two nodes is naturally defined as the length of the geodesic between the two nodes. Function `nx.eccentricity(G,v=Node)` returns the eccentricity for one node `v` or the whole graph. Note that in a directed graph, there may be no directed geodesics for some pairs of nodes. You must decide if it is appropriate to coerce the digraph to an undirected graph.

```
ecc = nx.eccentricity(nx.Graph(G))
print(ecc[ego])
```

```
❮ 3
```

The remaining "circular" network properties are defined through the eccentricity. If you already calculated it, do not throw it away, but pass to the following functions for the sake of performance.

- The diameter of a network is the maximum eccentricity. If two nodes are as far apart as possible, they must be at the diametrically opposite ends of the network, right?

- The radius of a network is the minimum eccentricity. This definition is not intuitive, but it is what it is. What's more counterintuitive, in general, is that the radius is not a half of the diameter.

- The center of a network is a set of all nodes whose eccentricity equals the radius. Another not very intuitive definition—but it yields a surprisingly accurate result (see the following example).

- The periphery of a network is a set of all nodes whose eccentricity equals the diameter. The set of peripheral nodes in a complex network is usually large.

In the following examples, all circular measures are calculated based on the precomputed eccentricity. There is no need to transform the digraph into a directed graph anymore.

```
print(nx.diameter(G, ecc))
```

```
4
```

```
print(nx.radius(G, ecc))
```

```
2
```

```
nx.center(G, e=ecc)
# Bingo!
```

```
['Complex Network']
```

```
nx.periphery(G, e=ecc)
['Glossary Of Areas Of Mathematics', 'Nutrition', 'Domestic Technology',
 «...», # 2,869 nodes!
 'Pierre Bourdieu', 'Sociology Of Law', 'Network Scheduler']
```

The eccentricity is a special case of path-based centralities: measures that discriminate nodes by their position in the network. Centralities are quintessential for social network analysis and most types of CNA in general.

Choose the Right Centralities

This section uses Matplotlib, Pandas, NumPy.

One of the goals of social network analysis is to identify actors with outstanding properties: the most influential, the most efficient, the most irreplaceable—in other words, the most important. CNA, in general, is also looking at the most important nodes: key products in product networks; key words in semantic networks; key events in the networks of events, and the like. One of the central premises of CNA is that the importance of a node depends on the structural position of the node in the network and can be calculated from neighborhoods, geodesics, or some other structural elements. Let's go over some of the most common centrality measures, without going deep into the theory. (If you're a curious reader, treat yourself with *Social and Economic Networks [Jac08]*, which covers the theory of centralities and many other CNA topics!)

Degree Centrality

The simplest centrality measure is a node degree (also indegree and outdegree, whenever necessary). Intuitively, a node with more edges, representing, say, an actor with more ties, is more important than a node with only one edge. Degree centrality is local and depends only on the node neighborhood. You saw this centrality in disguise in *Truncate the Network*, on page 46.

You may argue that the node with the largest degree in a small network may have fewer edges than the node with the smallest degree in a huge network. To

level the playing field, divide the number of edges by the maximal possible number of incident edges. Remember that in a network with a simple graph (no loops, no parallel edges), a node can have at most `len(G)-1` neighbors. The redefined degree centrality is always in the range from 0 (the node has no neighbors) to 1 (the node is the hub of the global star). The normalization makes it possible to compare nodes from different networks. The subject with the highest degree centrality in our Wikipedia network is *Computer Network* (0.227988).

A node with a high degree centrality may be capable of affecting a lot of neighbors in its neighborhood at once, but we cannot say anything about the opportunities for global outreach.

Closeness and Harmonic Closeness Centrality

The closeness centrality is defined as the reciprocal mean distance (length of the geodesics) from a node to all other reachable nodes in the network. It shows how close the node is to the rest of the graph. This centrality is also in the range from 0 (the node has no neighbors; it is severed from the rest of the network) to 1 (the node is the hub of the global star and is one hop away from any other node).

Another way to quantify the sense of closeness is to look at the mean reciprocal distance (as opposed to the reciprocal mean distance; the order of the sum and reciprocal operations reverses). Such measure is called harmonic centrality. Regrettably, the `NetworkX` function for calculating harmonic centrality does not normalize the result. Make sure you divide it by `len(G)-1` to obtain comparable measures.

When the closeness of a node is equal to 0 or 1, the harmonic closeness of the same node is 0 or 1, too. However, the two centralities in general differ and in the case of our Wikipedia network are not even strongly correlated. The subject with the highest closeness centrality is *Computer Network* (0.517678); *Graph (Discrete Mathematics)* performs best in terms of harmonic closeness (0.027257).

A node with a high degree centrality may be capable of affecting the entire network, but how about controlling it?

Betweenness Centrality

The betweenness centrality is for control freaks. It measures the fraction of all possible geodesics that pass through a node. If the betweenness is high, the node is potentially a crucial go-between (thus the name) and has a brokerage capability. The removal of such a node would disrupt communications

in communication networks, lengthen geodesics, lower closeness centralities, and possibly split the network into disconnected components.

The subject with the highest betweenness centrality in our Wikipedia network is *Computer Science* (0.005515).

You can often find high-betweenness nodes in the vicinity of bridges. A bridge is an edge whose removal would disconnect the network or significantly increase the length of the geodesics. The latter kind of bridge is called a *local bridge*. Pure bridges are rare in complex networks, but local bridges are not.

Eigenvector Centrality

Unlike the previously introduced centrality measures that rely on the neighborhoods and geodesics to calculate the importance, the eigenvector centrality uses a recursive definition of it: "Tell me who your friends are, and I will tell you who you are." (Incidentally, the saying can be traced back to Proverbs 13:20: "He that walketh with wise men shall be wise: but a companion of fools shall be destroyed.") Mathematically, the eigenvector centrality of a node is the sum of the neighbors' eigenvector centralities divided by λ—the largest eigenvalue of the adjacency matrix of the network.

High eigenvector centrality identifies nodes that are surrounded by other nodes with high eigenvector centrality. You can use this measure to locate groups of interconnected nodes with high prestige.

The subject with the highest eigenvector centrality in our Wikipedia network is *Graph (Discrete Mathematics)* (0.183307).

PageRank

At least two more types of centralities are based on recursive principles similar to the eigenvector centrality: *PageRank and HITS [PBMW99]*.

PageRank was developed by Google (and named after Google's Larry Page) to rank web pages. The web pages are represented by nodes in a directed graph. The graph edges correspond to hyperlinks. The rank of a node (and the corresponding page) in the network is calculated as the probability that a person randomly traversing the edges (clicking on links) will arrive at the node (page). The algorithm is parametrized by the damping factor `alpha=0.85`, which is the probability that the user will continue clicking. The page with the highest PageRank is the most attractive: no matter where the person starts, this page is the most likely final destination.

Markov Page

If you're familiar with Markov chains, you may have noticed that PageRank with `alpha=1` treats the network as a Markov chain and calculates its stationary distribution. The damping factor introduces an element of realism in the computation: the Web is dynamic and does not have a stationary state.

PageRank thrives on the concept of link traversing and makes sense only in directed networks. If you pass an undirected graph to `nx.pagerank()`, the function will first convert it into a directed graph by replacing each undirected edge with a pair of directed edges. The subject with the highest PageRank in our Wikipedia network is *Graph (Discrete Mathematics)* (0.000836).

HITS Hubs and Authorities

The HITS (Hyperlink-Induced Topic Search) algorithm is an extended version of PageRank. PageRank considers all graph nodes as potential terminals, or "sinks." Once you get into a sink, you likely get sunk. "Sink-style" networks include the Web, trust networks (social networks built on the "A-trusts-B" relationship), and organizational networks ("A-is-a-subordinate-of-B").

You want to study a network from the opposite perspective: what is the probability that a person randomly traversing the edges has *started* at the node? You can either reverse the graph by calling `G.reverse()` and then calculate the PageRanks—or execute the HITS algorithm and get both hubs and authorities values. Authorities are a loose counterpart of the PageRank. Hubs considers outgoing links instead of incoming links. They serve as entry points into your network so that you (or the fictitious randomly traversing person) could get to the authorities most efficiently.

The subjects with the highest hubs and authorities in our Wikipedia network are *Social Network* (0.037699) and *Graph (Discrete Mathematics)* (0.005213), respectively.

Comparing the Centralities

As Ulrik Brandes from University of Konstanz mentioned in his keynote address at the International Conference on Computational Social Science in July 2017, "There are several hundred centrality indexes." You just learned about seven or eight centrality measures (depending on whether to count the HITS as one or two), which may be 1 percent of all possible ways to establish a numeric order in a network. Do you have to learn about the remaining 99 percent?

No, you don't. First, those eight centralities adequately rank the nodes in any complex network. Second, even some of those eight centralities are strongly correlated. Let's find out which.

This next code calculates eight types of centralities for each node in the Wikipedia page network. Each function (except for nx.hits()) returns a dictionary with nodes as keys and centralities as values. nx.hits() returns a list of two dictionaries.

measuring.py

```
dgr = nx.degree_centrality(G)
clo = nx.closeness_centrality(G)
har = nx.harmonic_centrality(G)
eig = nx.eigenvector_centrality(G)
bet = nx.betweenness_centrality(G)
pgr = nx.pagerank(G)
hits = nx.hits(G)

centralities = pd.concat(
    [pd.Series(c) for c in (hits[1], eig, pgr, har, clo, hits[0], dgr, bet)],
    axis=1)

centralities.columns = ("Authorities", "Eigenvector", "PageRank",
                        "Harmonic Closeness", "Closeness", "Hubs",
                        "Degree", "Betweenness")
centralities["Harmonic Closeness"] /= centralities.shape[0]
```

Then comes Pandas (check *Another Kind of Pandas,* on page 73). Let's convert each dictionary into a pd.Series—a labeled vector; then concatenate all vectors into a pd.DataFrame—a labeled matrix. Finally, relabel the columns to reflect their true nature and normalize the harmonic closeness centrality, because it is the only centrality on the list that is not automatically normalized. The DataFrame centralities is ready for analysis.

centralities.corr() calculates all pairwise correlations between the centralities and returns an 8×8 symmetric DataFrame. More than half of the values in the DataFrame are redundant. Use np.tri() from NumPy to generate a lower-left triangular unit matrix of the proper size and mask the duplicates by multiplying the DataFrame and the mask matrix element-wise.

Locating the strongest correlations in the table may be hard. Let's reorganize it into a tall pd.Series, sort by the values, and display the "tail"—the last five rows.

measuring.py

```
# Calculate the correlations for each pair of centralities
c_df = centralities.corr()
ll_triangle = np.tri(c_df.shape[0], k=-1)
c_df *= ll_triangle
c_series = c_df.stack().sort_values()
c_series.tail()
```

```
Harmonic Closeness  Eigenvector    0.826834
Closeness           Authorities    0.828464
Betweenness         Degree         0.849580
PageRank            Eigenvector    0.882332
Eigenvector         Authorities    0.939547
dtype: float64
```

The complete analysis of all correlations (which you can do on your own, especially after reading *Outline Modularity-Based Communities*, on page 136) reveals that the centrality measures form two groups. The first group consists of eigenvector and harmonic closeness centralities, PageRank, and authorities. The second group has two subgroups: degree and betweenness centralities in one, and closeness and hubs in the other. I am almost saying that knowing one representative measure from each group—say, closeness, betweenness, and eigenvector centralities—probably will suffice for all practical purposes. But the final choice is yours.

To add another dimension to our story of centralities, let's plot one of them against another. Pandas DataFrames are elegantly integrated with Matplotlib. It takes just a couple of function calls to plot two columns.

measuring.py

```
X = "Harmonic Closeness"
Y = "Eigenvector"
limits = pd.concat([centralities[[X, Y]].min(),
                    centralities[[X, Y]].max()], axis=1).values
centralities.plot(kind="scatter", x=X, y=Y, xlim=limits[0], ylim=limits[1],
                  s=75, logy=True, alpha=0.6)
```

The figure on page 98 shows the scatter plot of harmonic closeness and eigenvector centrality for our network of Wikipedia pages. Note that the vertical axis has a logarithmic scale to accommodate small eigenvector centralities. Judging by the plot, the correlation of 0.826834 is entirely justifiable.

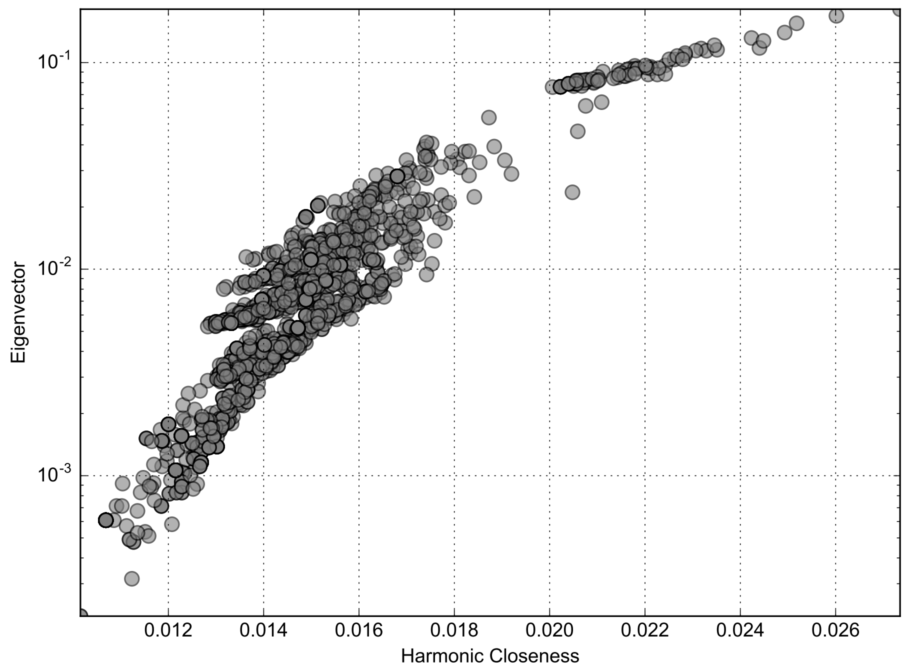
Estimate Network Uniformity Through Assortativity

This section uses NumPy.

In the last section of the chapter, let's look at node attributes we have completely ignored so far. As an example, I'll use a snapshot of *Odnoklassniki*,[2] the second-largest Russian language social networking site, harvested in February 2009. The snapshot has 408,715 nodes and 4,482,086 edges. Each node has attributes *age* and *gender* (self-reported).

Attribute analysis looks into assortativity: correlation between the values of a node attribute across edges. A network with positively correlated attributes

2. ok.ru



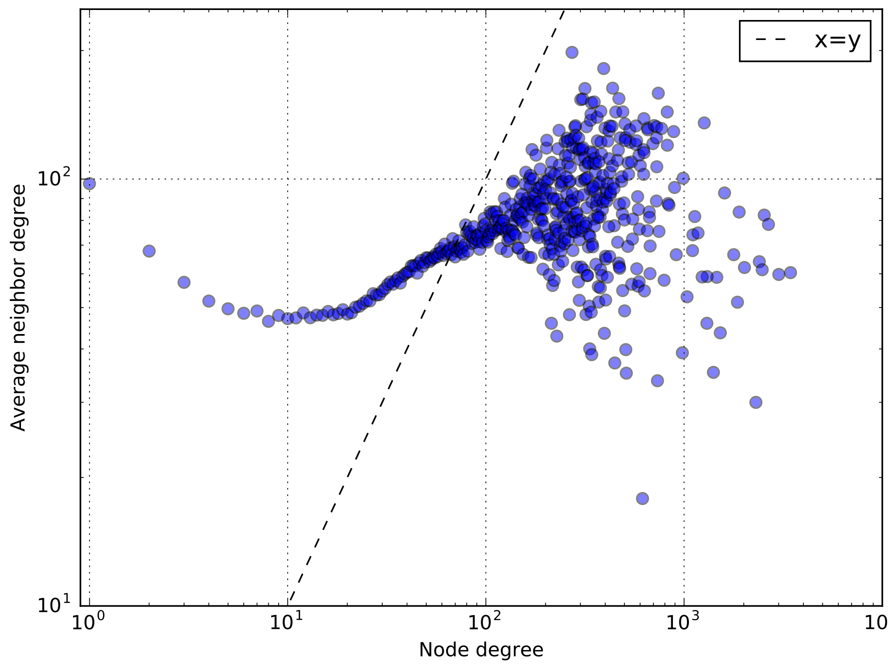
is called assortative; in an assortative network, nodes tend to connect to nodes with similar attribute values. This tendency is called assortative mixing. A dissortative (negatively correlated) network is the opposite of an assortative one.

The simplest form of assortativity is degree (indegree, outdegree) assortativity: the correlation between the degree of a node and the average degree of its neighbors. Function `nx.average_degree_connectivity(G)` returns a dictionary with unique node degrees as keys and matching average neighbors' degrees as values. The following code fragment calculates the dictionary and separates the keys and values into two lists (`my_degree` and `their_degree`):

```
my_degree, their_degree = zip(*nx.average_degree_connectivity(G).items())
```

The figure on page 99 shows the scatter plot of the two lists for the *Odnoklassniki* network. If the network were uniform, all points would align along the dashed line. In reality, we observe the uniformity only around the nodes with about seventy neighbors. Nonetheless, the network is in general assortative because the slope of the curve is positive. The only dissortative part of the network is the one that contains the nodes with fewer than ten edges.

Degree assortativity may be somewhat hard to interpret, but when it comes to other attributes, especially related to human demographics, there are certain



expectations—and a social theory that explains them: homophily. Homophily is the propensity of actors to associate with somewhat similar actors. Let's check if our fragment of the social network is assortative with respect to age and gender.

NetworkX provides two functions for assessing attribute assortativity. The first function nx.attribute_mixing_matrix() takes a graph, an attribute name, and an optional mapping dictionary, and returns a two-dimensional NumPy array.

```
nx.attribute_mixing_matrix(G, "gender", mapping={"M": 0, "F": 1})
```

```
array([[ 0.22771058,  0.24064205],
       [ 0.24064205,  0.29100532]])
```

The ith row and jth column of the array contain the fraction of adjacent nodes that have the ith and jth values of the attribute, respectively. The mapping links non-numeric attribute values with row and column indexes. In the previous matrix, 0.22771058 (≈23%) of edges connect male actors, and 0.29100532 (≈29%) of edges connect female actors. The fraction of same-gender edges is just above 50 percent, which suggests that the members of the network do not prefer same-gender connections. The network is not homophilic from the gender point of view.

The second function, nx.attribute_assortativity_coefficient(), confirms the previous result. The function returns the assortativity coefficient—the correlation between the values of an attribute across edges.

```
nx.attribute_assortativity_coefficient(G, "gender")
```

```
❮ 0.03356000539110733
```

The self-reported genders of the *Odnoklassniki* members are not correlated. However, their ages are in a better agreement:

```
nx.attribute_assortativity_coefficient(G, "age")
```

```
❮ 0.14409535867553133
```

The last result is not surprising at all if we recall that *odnoklassniki* in Russian means *classmates* and it is natural for classmates to be of the same age.

You learned how to measure a complex network by calculating its microscopic, mesoscopic, and macroscopic properties, such as size, density, clustering coefficient, centralities, and assortativities. The measured properties identify the most important or unusual nodes and network neighborhoods. With a couple dozen network measuring algorithms in your toolbox, you are ready for a complete network analysis experiment. In the next chapter, you will go through the first complete CNA case study.

This good luck of those of Chagre caused Captain Morgan to stay longer at Panama, ordering several new excursions into the country round about; and while the pirates at Panama were upon these expeditions, those at Chagre were busy in piracies on the North Sea.

➤ *Alexandre Olivier Exquemelin, French, Dutch or Flemish writer*

CHAPTER 9

Case Study: Panama Papers

This chapter uses Matplotlib, NumPy, Pandas.

The Panama Papers[1] represent a massive leak of offshore corporate entity information (several hundred thousand entities) from the Panamanian law firm *Mossack Fonseca*. The papers unveil a never-before-seen network of money-laundering connections.

In this chapter, you will learn how to convert a huge CSV file describing connections between entities and officials into a social network. You will do it two ways: with and without Pandas. You will also learn how to make simple conclusions about the resulting network.

Create a Network of Entities and Officers

The "Panama" network is a social network that describes relationships between organizations and individuals traced through electronic documentation. The network is available in five CSV files that is summarized in the table on page 102.

Let's first partially build this vast network and analyze some of its aspects without using Python's "heavy artillery," Pandas, and later attempt a similar analysis with Pandas. For the construction, let's select only the edges that refer to the "beneficiary-of" relationship (there are 19,194 edges labeled *Beneficiary of* and *beneficiary of*). We will go through the files Entities.csv, Officers.csv, and Intermediaries.csv in search of incident nodes. For each node, we will store its name, type, and a three-letter country code. The first code block imports all the necessary modules and defines the constants.

1. www.occrp.org/en/panamapapers/database

Name	Type	Purpose	# of rows	Columns of interest
all_edges.csv	Edges	Each edge has a type of the represented relationship.	1,269,796	*node_1*, *rel_type*, *node_2*
Addresses.csv	Nodes	Legal addresses of officers and entities	151,127	n/a
Entities.csv	Nodes	Legal entities (corporations, firms, and so on)	319,421	*name*, *jurisdiction*
Intermediaries.csv	Nodes	Persons and organizations that act as links between other organizations	23,642	*name*, *country_code*
Officers.csv	Nodes	Persons (directors, shareholders, and so on)	345,645	*name*, *country_code*

panama.py

```
import csv
import pickle
import itertools
from collections import Counter
import networkx as nx
from networkx.drawing.nx_agraph import graphviz_layout
import matplotlib.pyplot as plt
import dzcnapy_plotlib as dzcnapy

EDGES = "beneficiary"
NODES = (("Entities.csv", "jurisdiction", "name"),
         ("Officers.csv", "country_codes", "name"),
         ("Intermediaries.csv", "country_codes", "name"))
```

Where to Import?

According to PEP 8, Style Guide for Python Code, “imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.”[2] Some developers argue that importing a module immediately before its first use saves a nanosecond or so. It might, but it sure makes tracking code dependencies on other libraries a nightmare.

2. www.python.org/dev/peps/pep-0008/#imports

Now, it's time to build a network graph. Start with an empty nx.Graph object and read all rows from all_edges.csv with a CSV dictionary reader. (But keep only those future edges that are marked as *beneficiary of* in any character case.)

panama.py

```
panama = nx.Graph()

with open("all_edges.csv") as infile:
    data = csv.DictReader(infile)
    panama.add_edges_from((link["node_1"], link["node_2"])
                          for link in data
                          if link["rel_type"].lower().startswith(EDGES))
```

Remember that when you add an edge to a network, NetworkX also adds both incident nodes. However, at the moment, the nodes have only more or less randomly chosen labels—and no attributes. Let's import the attributes and true names from the other three files.

The purpose of the dictionary nodes is to facilitate future lookup. (Python lists have linear lookup time.) Read each of the files with a CSV dictionary reader and extract and collect the desired attributes. Note that there is no need to process rows that do not match any existing node (because your network does not include all nodes and edges) and add any nodes to the graph (because they have been already added by way of the incident edges). When done, update the node attributes *country* and *kind*, and relabel the nodes to match persons and organizations names.

panama.py

```
nodes = set(panama.nodes())
relabel = {}

for f, cc, name in NODES:
    with open(f) as infile:
        kind = f.split(".")[0]
        data = csv.DictReader(infile)
        names_countries = {node["node_id"] :
                           (node[name].strip().upper(), node[cc])
                           for node in data
                           if node["node_id"] in nodes}
    names =     {nid: values[0] for nid, values in names_countries.items()}
    countries = {nid: values[1] for nid, values in names_countries.items()}
    kinds =     {nid: kind      for nid, _      in names_countries.items()}
    nx.set_node_attributes(panama, "country", countries)
    nx.set_node_attributes(panama, "kind", kinds)
    relabel.update(names)

nx.relabel_nodes(panama, relabel, copy=False)

if "ISSUES OF:" in panama:
    panama.remove_node("ISSUES OF:")
```

```
if "" in panama:
    panama.remove_node("")

print(nx.number_of_nodes(panama), nx.number_of_edges(panama))
```

Finally, remove the phony node *ISSUES OF:* (there is no explanation of its purpose in the dataset documentation) and a mysterious node with no name at all. The complete network panama has 27,930 (3.3%) nodes and 19,137 (1.5%) edges. The density of the network is 0.000049.

As it turns out, the newly minted network consists of several thousand disconnected fragments called components, most of which have only two to three nodes. You will learn how to work with components in *Split Networks into Connected Components, on page 126*. However, looking ahead, you can use function nx.connected_component_subgraphs() to elicit each component's graph and keep it only if it has at least twenty edges or twenty nodes. The choice of numbers is somewhat arbitrary; you may want to play with them to wipe off as much of the "network dust" as you wish. In fact, you are under no obligation to do any filtering at all—or you can select only the biggest component.

panama.py

```
components = [p.nodes() for p in nx.connected_component_subgraphs(panama)
              if nx.number_of_nodes(p) >= 20
              or nx.number_of_edges(p) >= 20]
panama0 = panama.subgraph(itertools.chain.from_iterable(components))

print(nx.number_of_nodes(panama0), nx.number_of_edges(panama0))

with open("panama-beneficiary.pickle", "wb") as outfile:
    pickle.dump(panama0, outfile)
```

The refined network panama0 has 1,393 nodes and 1,926 edges. Pickle it for future use!

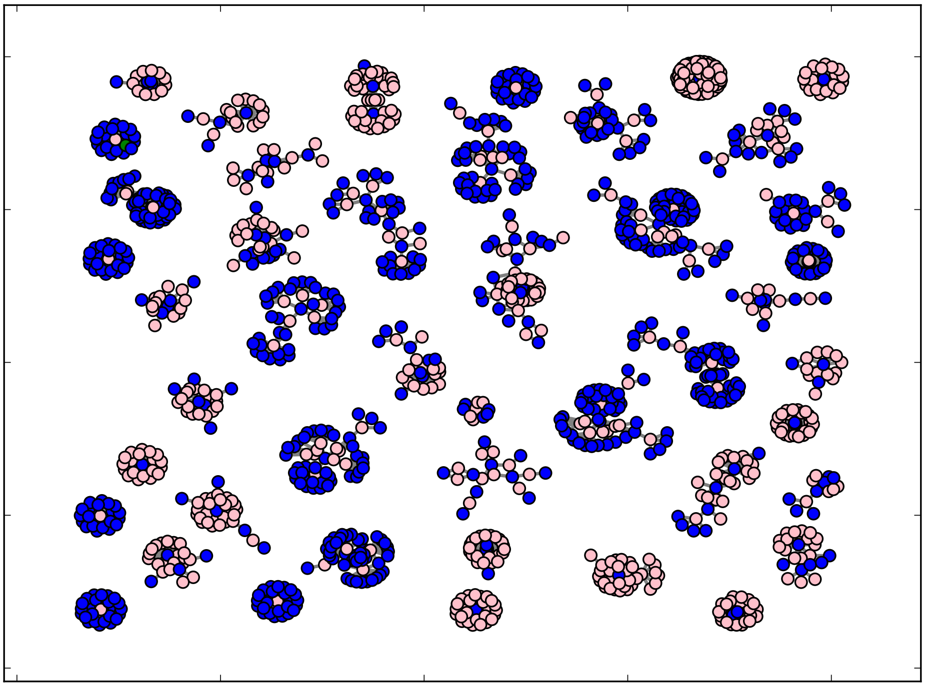
Draw the Network

The size of the network generated in the previous section makes its visualization almost useless. However, the plotting fragment is still included in the case study. Nodes are painted by their kind: Entities are lightly colored, and Officers are dark.

panama.py

```
cdict = {"Entities": "pink", "Officers": "blue",
        "Intermediaries" : "green"}
c = [cdict[panama0.node[n]["kind"]] for n in panama0]
dzcnapy.small_attrs["node_color"] = c
pos = graphviz_layout(panama0)
nx.draw_networkx(panama0, pos=pos, with_labels=False, **dzcnapy.small_attrs)
dzcnapy.set_extent(pos, plt)
dzcnapy.plot("panama0")
```

The following figure shows the sketch of the network (naturally, without the labels). You can see with a naked eye that some components have an Officer in the center, surrounded by Entities; in some other components, conversely, Officers surround an Entity in the center.



We can learn more about the network by applying numerical analytical methods to it.

Analyze the Network

Out of so many ways to analyze a network, let's focus on degree and assortativity analysis. All network nodes have two attributes: *kind* (with three possible values *Entities*, *Officers*, or *Intermediaries*) and *country*. Let's have a look at each of the assortativities, both directly and through the attribute mixing matrix (only for the *kind*).

panama.py

```
nx.attribute_assortativity_coefficient(panama0, "kind")
nx.attribute_mixing_matrix(panama0, "kind",
                           mapping={"Entities": 0, "Officers": 1,
                                    "Intermediaries" : 2})
nx.attribute_assortativity_coefficient(panama0, "country")
nx.degree_assortativity_coefficient(panama0)
```

```
-0.9896603076687625
array([[  0.00000000e+00,   4.97403946e-01,   2.59605400e-04],
       [  4.97403946e-01,   4.67289720e-03,   0.00000000e+00],
       [  2.59605400e-04,   0.00000000e+00,   0.00000000e+00]])
0.07539400284377736
-0.39717073403670283
```

The network is almost perfectly dissortative with respect to the node kinds. The matrix explains the details: an Entity node (column 0) is almost always connected to an Officer node (column 1) because Officers are "beneficiaries-of" Entities. Very few Entities are linked to each other through Intermediaries. The most obscure edges connect Officers to Officers. Without fully understanding the semantics of the relationship "beneficiary-of," you cannot judge whether these edges are legitimate or not. Nonetheless, the matrix gives you a generalized snapshot of how a typical network fragment would look.

The "Panama" network is rooted in corporate offshoring. You would expect that the country codes associated with network nodes are quite diverse because that is what offshoring is all about. And indeed they are: a correlation close to 0 (0.075) is no correlation. The network of country codes is neither assortative nor dissortative. It has random connectivity, totally appropriate for offshoring-based money laundering.

The last number in the output is the degree assortativity coefficient. It is negative, suggesting that the nodes with a higher degree are surrounded, on average, by nodes with a smaller degree, and the other way around.

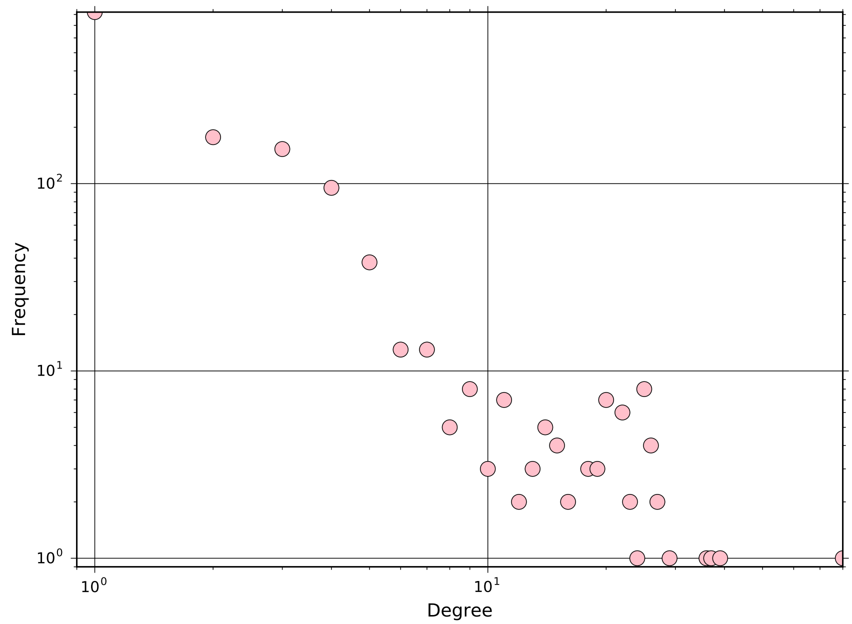
An essential output of complex network analysis is a node degree distribution. According to *Barabási and Albert [BA99]*, if a network is a result of preferential attachment, then the degrees `d` in the network are distributed by the power law: `p(d)=d`$^{-a}$. The converse, in general, is not true. (See a brief overview on page 5 and more in *What Makes Components Giant?*, on page 129.) If you apply `log()` to the equation, you will get a linear dependency: `log(p(d))=-a log(d)`.

Let's check if the network has at least a chance of being a Barabási–Albert network. The next code fragment calculates the degrees of all nodes, and counts and plots the frequency of each degree.

panama.py

```
deg = nx.degree(panama0)
x, y = zip(*Counter(deg.values()).items())
```

The degree distribution in the log-log scale is in the figure on page 107. The dots align along a noisy but clearly visible straight line, signaling you that the "Panama" network, like many other social networks, may have been put in order by the forces of preferential attachment.



One peculiarity of a power law distribution is its "long tail" that, at least theoretically, tolerates nodes with arbitrary high degrees, limited only by the total graph size. Your degree distribution has a long tail, too. So, who are they, the nodes at the tail? Let's run the final lines of the analysis script that reports the top ten nodes with the highest degree, nicely formatted.

panama.py

```
top10 = sorted([(n, panama0.node[n]["kind"], v) for n, v in deg.items()],
               key=lambda x: x[2], reverse=True)[:10]
print("\n".join(["{} ({}): {}".format(*t) for t in top10]))
```

```
HELITING S.A. (Officers): 80
T.K.B.K. INTERNATIONAL TRUST (Entities): 39
WORLDWIDE COM-NET INTERNATIONAL TRUST (Entities): 37
THE CLAUDIUS TRUST (Entities): 36
GUANGZHOU CONSTRUCTION & DEVELOPMENT HOLDINGS (CHINA)LIMITED (Officers): 29
RICARDO CAMPOLLO CODINA (Officers): 27
ISLANDS INTERNATIONAL TRUST (Entities): 27
ZEN TRUST (Entities): 26
FRIENDS OF ASSISI TRUST (Entities): 26
MR. OLEKSII MYKOLAYOVYCH AZAROV (Officers): 26
```

Four of the tail nodes are Officers; the other five are Entities. The Intermediaries node did not make it to the top list. I don't know who these individuals and organizations are. Chances are, you don't know, either. In that case, just include the results into your final report and deliver to your data sponsor—someone who ordered the analysis of this complex network.

Build a "Panama" Network with Pandas

Pandas is a library that is known to make hard tasks of reading tabular files and manipulating tabular data easy. Pandas takes care of parsing an edge list from a CSV file into a DataFrame; nx.from_pandas_dataframe() converts a DataFrame into a graph, making network construction all but trivial.

Now let's take care of the nodes. Read the three node attribute files into their DataFrames, mark properly (so that you would know later which nodes come from each file), and merge the parts into one DataFrame named all_nodes.

panama-ca.py

```
import networkx as nx
import pandas as pd
import numpy as np

# Read the edge list and convert it to a network
edges = pd.read_csv("all_edges.csv")
edges = edges[edges["rel_type"] != "registered address"]
F = nx.from_pandas_dataframe(edges, "node_1", "node_2")

# Read node lists
officers = pd.read_csv("Officers.csv", index_col="node_id")
intermediaries = pd.read_csv("Intermediaries.csv", index_col="node_id")
entities = pd.read_csv("Entities.csv", index_col="node_id")

# Combine the node lists into one dataframe
officers["type"] = "officer"
intermediaries["type"] = "intermediary"
entities["type"] = "entity"

all_nodes = pd.concat([officers, intermediaries, entities])
```

Just like any other real-life data, the "Panama papers" dataset is "dirty": it has duplicates, omissions, typos, and so on. With the following code fragment, you can partially unify the duplicated names by removing all leading and trailing whitespaces, merging all inner whitespaces, converting all characters to the uppercase, converting *LIMITED* to *LTD*, and removing some honorifics.

panama-ca.py

```
# Do some cleanup of names
all_nodes["name"] = all_nodes["name"].str.upper().str.strip()

# Ensure that all "Bearers" do not become a single node
all_nodes["name"].replace(
    to_replace=[r"MRS?\.\s+", r"\.", r"\s+", "LIMITED", "THE BEARER",
                "BEARER", "BEARER 1", "EL PORTADOR", "AL PORTADOR"],
    value=["", "", " ", "LTD", np.nan, np.nan, np.nan, np.nan, np.nan],
    inplace=True, regex=True)
```

A lot of “Panama” officials go under the nicknames “THE BEARER” and “EL PORTADOR.” If left unchanged, they may be later lumped into a single network node. Rename them to a NumPy value np.nan to keep them anonymous but distinct.

The network is structurally ready, but the nodes do not have attributes—the attributes are stored in a separate DataFrame. Attaching them to the nodes now is impractical. It is highly unlikely that you will analyze the whole network at once, so it makes no sense to invest into decorating all the nodes. Let’s first identify the area of interest, extract it from the network, and then assign the attributes and new labels to the surviving nodes. This order of network construction is contrary to anything you’ve seen so far, but for a big network, it saves you a lot of CPU time and computer memory.

As an exercise, extract a network of officers, entities, and intermediaries related to the economically, politically, and geographically compact region—Central Asia (Kazakhstan, Kyrgyzstan, Uzbekistan, Turkmenistan, and Tajikistan). Offshoring, especially fraudulent offshoring, is not very widespread in this area, and we can hope to obtain a reasonably small subnetwork.

Start with a list of seed nodes seeds—all nodes that are known to belong to the Central Asian region because their country codes are in the set of wanted country codes CCODES.

In reality, some nodes have more than one country code, in which case the codes are separated by a semicolon. Can you modify the code to select the nodes that are associated with Central Asia through *at least one* of their country codes?

What you will do next is essentially construct a joint ego network of the seed nodes expanding two hops away from the seeds. The function nx.single_source_shortest_path_length(F,seed,cutoff=None) computes the shortest paths from the node seed to all reachable nodes that are cutoff hops away and closer. The function returns a dictionary with the target nodes as keys, so the keys are the cutoff-neighborhood of the seed. Extract a subgraph that contains all the keys for all the dictionaries with all the connecting edges. It is the network that you want.

panama-ca.py

```
CCODES = "UZB", "TKM", "KAZ", "KGZ", "TJK"
seeds = all_nodes[all_nodes["country_codes"].isin(CCODES)].index
nodes_of_interest = set.union(*[\
        set(nx.single_source_shortest_path_length(F, seed, cutoff=2).keys())
        for seed in seeds])

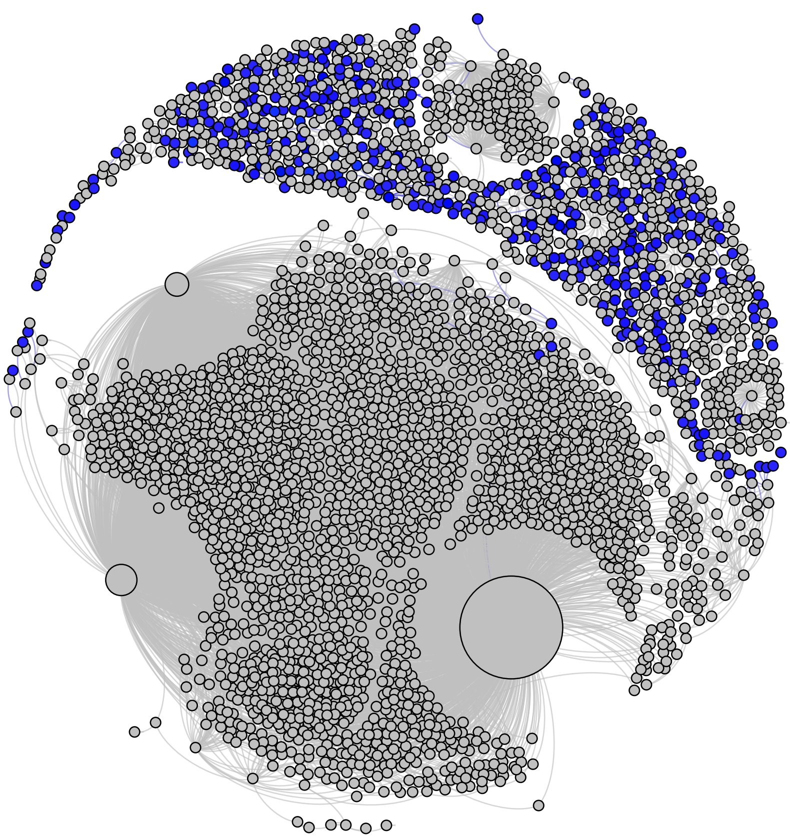
# Extract the subgraph and relabel it
ego = nx.subgraph(F, nodes_of_interest)
```

```
nodes = all_nodes.ix[ego]
nodes = nodes[~nodes.index.duplicated()]
nx.set_node_attributes(ego, "cc", nodes["country_codes"])
valid_names = nodes[nodes["name"].notnull()]["name"].to_dict()
nx.relabel_nodes(ego, valid_names, copy=False)

# Save and proceed to Gephi
with open("panama-ca.graphml", "wb") as ofile:
    nx.write_graphml(ego, ofile)
```

The subnetwork has 3,848 nodes and 8,643 edges—quite modest, by the CNA standards. Because of the imperfection of the dataset, some nodes may have identical identifiers—remove them (see the highlighted line). Finally, set the *cc* (country code) attribute and relabel the nodes that can be relabeled.

You can save the resulting network into a .graphml file for future use. For instance, you can sketch the network in Gephi, as shown in the following figure. (NetworkX itself is not powerful enough to produce images of big networks.)



The dark (blue) nodes represent the officers, entities, and intermediaries related to the Central Asian countries. The gray nodes are in their 2-neighborhood. The node size represents degree. You can see that the dark nodes congregate on the periphery of the network, despite formally being its seeds! The major offshoring companies—ShareCorp Ltd., Portcullis Trustnet (BVI) Ltd., and Execorp Ltd. (the three largest circles in the chart)—do not seem to be very interested in Central Asia.

You can apply the method presented in the case study to any network of organizations (entities) and members (officials). Beware that if the network is large, it is better to avoid `NetworkX` for its visualization and use `Gephi`.

In the Next Part

It is exciting to analyze networks with explicitly connected nodes. It is even more exciting to analyze networks with no immediate connections. The next part of the book looks into co-occurrence networks.

Part III

Networks Based on Co-Occurrences

An interesting, relatively understudied (compared to social networks), and crucial class of complex networks is networks based on item co-occurrence—items being in the same place (or close enough) at the same time. In this part, you will learn how to construct "unorthodox" networks and analyze their structure.

Life is, of course, a series of coincidences, but we never cease to be surprised as each new one happens, and nothing can destroy their recurring freshness.

➤ *Robert Lynd, Irish writer*

CHAPTER 10

Constructing Semantic and Product Networks

An interesting, novel, and relatively understudied class of complex networks is networks based on co-occurrence, or coincidence—the property of items being in the same place (or close enough) at the same time. The edges in co-occurrence networks are implicit: they are not given (and often not even obvious); you have to deduce, extract, and calculate them from other data, and this is a significant departure from the relatively intuitive way you build social networks. Co-occurrence networks are living proof that you can connect anything to anything and make sense of the connections.

In this chapter, you will learn how to start with a seemingly odd collection of material or immaterial items, examine temporal and spatial connections between them, identify significant relationships, and convert your observations into a network graph. Just like social network graphs, these graphs have nodes and edges with respective attributes, but this time you will go one step further and explore their complex internal structure. You will be able to divide and conquer a complex network: decompose it into components, cores, coronas, communities, and similar structural elements; assign proper names to the extracted parts; understand their purpose and importance; and put them together again.

We will start by looking at two examples of co-occurrence networks: semantic networks and product networks. In the next chapters, we'll go over definition, extraction, naming, and use of complex network constituents.

Semantic Networks

A semantic network is a network of nodes that represent terms—words, word stems, word groups, or concepts—connected based on the similarity or dissimilarity of their usage or meanings. Link terms that:

- Are commonly used together in the same place in text: same sentence, paragraph, chapter, scene, act, list of keywords, list of interests in a social network, and so on ("semantic" ↔ "network")
- Describe the same property ("red" ↔ "blue")
- Occupy the same semantic niche (synonyms: "program" ↔ "application"; hypernyms: "pet" ↔ "cat"; antonyms: "erase" ↔ "restore")

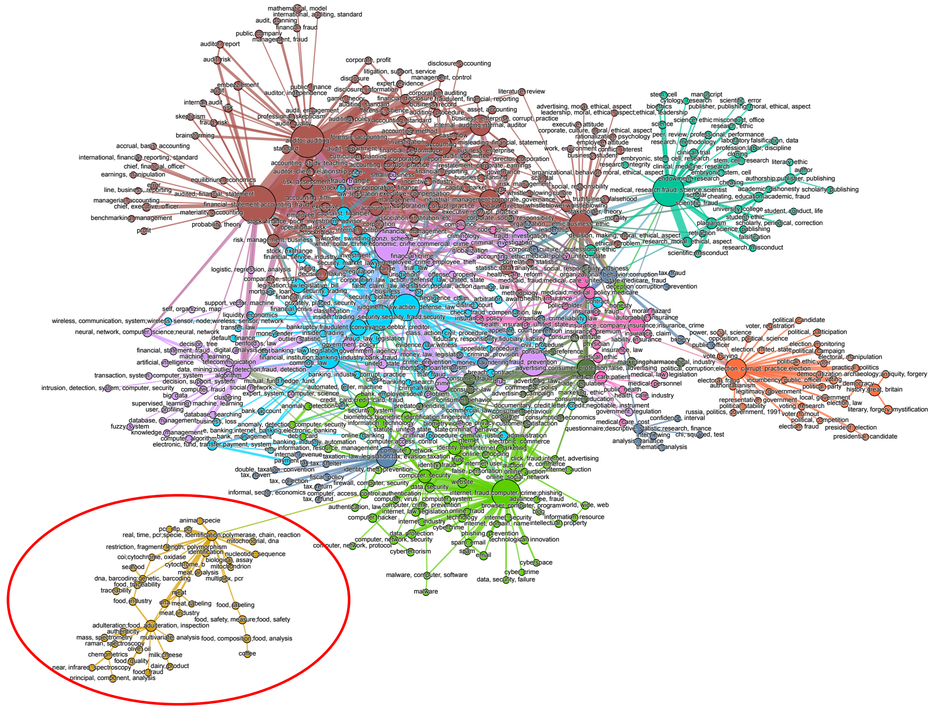
In the latter case, you may want to assign negative weights to some edges, which would make many network processing algorithms heartbroken. If your network has negatively weighted edges by construction, be prepared to remove them before analyzing the network.

Knowledge specialists use semantic networks for graphical (and machine-readable) knowledge representation, and social and behavioral researchers and anthropologists use semantic networks for semantic domain analysis. Let's have a look at two not-so-typical semantic networks: a network of keywords for fraud-related research papers and a network of characters from *Othello*.

Detect Food Fraud

Semantic networks often reveal surprising facts about texts and other term collections (corpora). Suppose you do research in accounting—namely, in fraud—and want to know everything about fraud types. You understand that nobody knows fraud better than other fraud researchers and fraudsters themselves. The latter are typically off limits, but the former are well represented in numerous databases of academic research papers. You could collect all research papers that mention "fraud," extract subject tags assigned to them by database editors, and create a semantic network of the tags, based on their co-occurrence. The subject tags (such as *DNA* and *meat industry*) are the nodes of the network. Two tag nodes are adjacent if the tags are frequently assigned together to the same paper. For example, the nodes *food fraud* and *food safety* are adjacent because many research papers focus on food fraud *and* food safety.

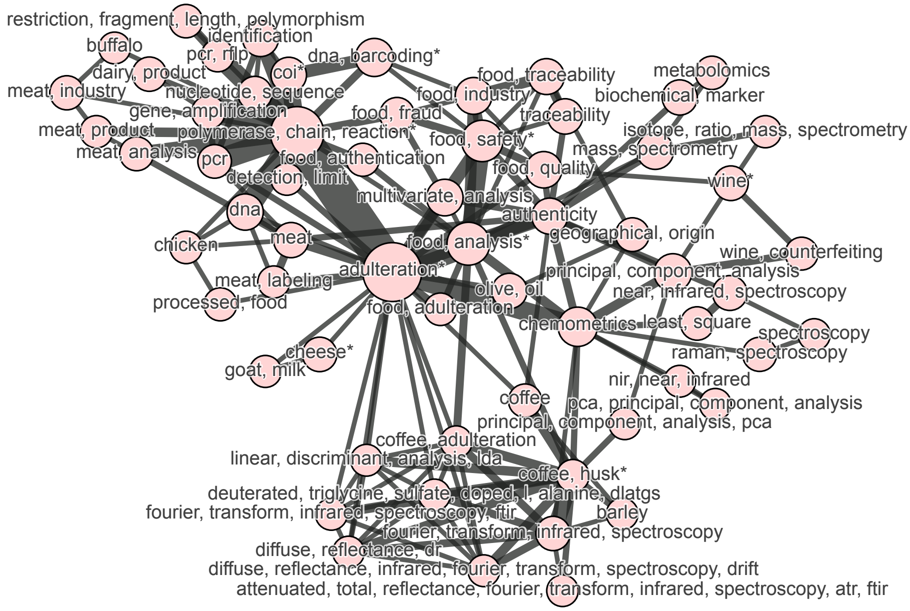
The original network (adapted from *Conceptual Structure of Fraud Research and Its Dynamics [GZ17]*), shown in the figure on page 117, is huge and could tell us many an exciting story. However, we will look only at the circled fragment in the bottom left corner.



The most striking conclusion from the figure is that the selected fragment is almost entirely removed from the rest of the network. It is connected to the bulk of the network with only one edge.

The figure on page 118 depicts the close-up view of the fragment. Remember that as a rule, node size represents node importance (in our case, the number of tags), and edge width represents edge weight. The nodes are colored based on their membership in network communities (*Outline Modularity-Based Communities, on page 136*), but this property is not relevant now.

Just by glancing through the node labels, you can see that the topic of the fragment is food fraud, also known as adulteration (no connection to adultery, adult stores, or any other "adult" business). Apparently, there is "fraud," and there is "food fraud!" Within the "food fraud" fragment, you can see tags related to fraud objects ("milk," "olive oil," "meat"); fraud detection methods ("spectroscopy," "DNA," "principal component analysis"); fraud prevention mechanisms ("food labeling," "identification"), and so on. If you are a PhD student or young postdoc looking for a future fraud-related research direction, you may be excited to have come across this semantic network fragment. Judging by its secluded position, few "hardcore" fraud analysts know or care about food! Why not become one of those who do?



By the way, we will show you how to construct a similar network step-by-step in Chapter 12, *Case Study: Performing Cultural Domain Analysis*, on page 141.

Expose a Protagonist

Who is the protagonist (or the main character, if you prefer less academic speak) of *Othello*? Be careful with what you answer because a trivial question like this must be tricky. Hint: No, Othello is not a protagonist. At least not by the standards of semantic networks.

The emerging field of digital humanities uses co-occurrence semantic networks to analyze texts: plays, scripts, and other forms of prose and poetry. The method allows us to identify the main and peripheral characters (see core-periphery analysis on page 129); group characters and places (see *Outline Modularity-Based Communities*, on page 136); and eventually break down the storyline into scenes suitable, say, for film or stage adaptation.

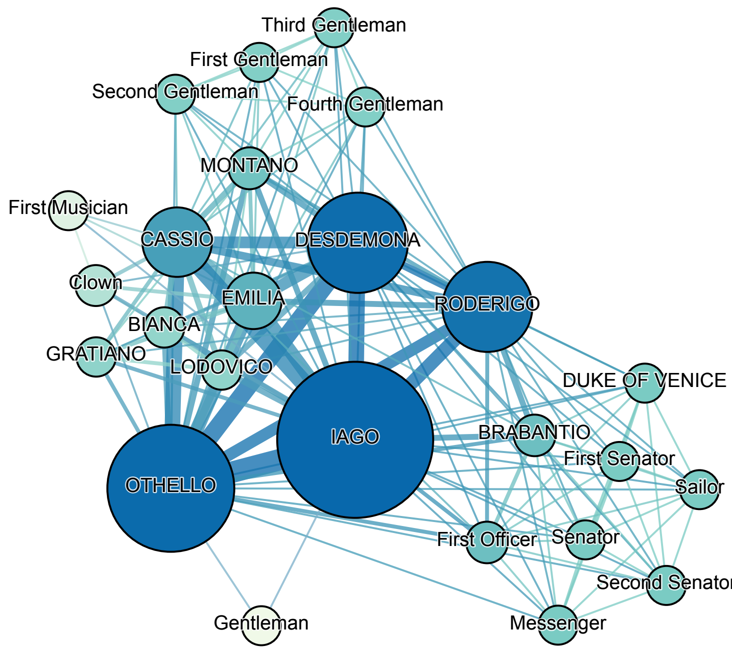
Let's outline a semantic network construction from the text of *Othello*. After you read the next chapter and the case studies, you will be able to implement the algorithm in Python. This exercise is inspired by *Measuring Tie Strength in Implicit Social Network [EG12]*.

1. You need a list of all characters. *Othello* is a short text; you can compose the list by hand. Alternatively, find all references to *Enter* and *Exit* remarks; or collect references to all characters as they speak if there is a property

in the text that identifies the characters. For example, a character may be marked with an HTML tag, as in `<A NAME=speech1><b>RODERIGO</b></a>`.[1]

2. You need a definition of co-occurrence. Play scripts are perfect from this point of view: two characters co-occur if they occur in the same scene! In a general text, co-occurrence may be based on paragraphs, sections, chapters, pages, and so on.

3. Now that you have characters (nodes) and their co-occurrences (edges), you can build a network. Remarkably, once constructed, this network is a social network, of which you heard so much in Chapter 6, *Understanding Social Networks*, on page 53. The result is shown in the following figure.



4. Finally, you need a measure of importance. How do you know, indeed, who is the protagonist of the story? Luckily, you have the whole box of network centralities (*Choose the Right Centralities*, on page 92) that you can apply to each node. When you work with a social network, and the network in the figure *is* a social one, the best importance measures are betweenness and eigenvector centralities. The eigenvector centrality is proportional to the graph node sizes, and the betweenness centrality is reflected by the node color (the darker, the more central). Both centralities seem to be in good agreement: Iago is the protagonist. Not Othello.

1. shakespeare.mit.edu/othello/full.html

So, is Iago indeed the protagonist of *Othello*? Some researchers strongly believe that he is![2] Welcome to digital humanities!

Product Networks

A product network is a network of retail items. Network nodes in a product network represent items purchased by individuals and co-occurring in their shopping baskets or carts. You can connect two product nodes if customers often or always buy the respective products together. We call such products complements. Left and right shoes (if sold separately), nuts and bolts, nails and hammers, and one-way airline tickets from Boston to Seattle and from Seattle to Boston are good examples of complements: when you buy one, you almost always buy the other as well.

Product networks can (but do not have to) be weighted: you can define the weight of the edge as the frequency of co-purchasing. You can slice (*Slice Weighted Networks*, on page 79) the network later to remove low-weighted edges, if you want.

Sometimes product networks allow negatively weighted edges. If one of the products in a pair is a reasonable replacement for the other—in some sense!—we call them substitutes. If you live in Alaska and buy a husky to pull your sled, then you probably won't buy a reindeer for the same purpose, at least not at the same time. (You can still get a reindeer as a pet.) A husky and reindeer are substitutes; you can connect the respective nodes with a negatively weighted edge to represent their substitutive nature.

Here are two product networks for you, as a warm-up: a network of common cooked food ingredients and a network of tools and materials for a painting do-it-yourself project.

Explore Your Pantry

To find a product network, look no further than your pantry.

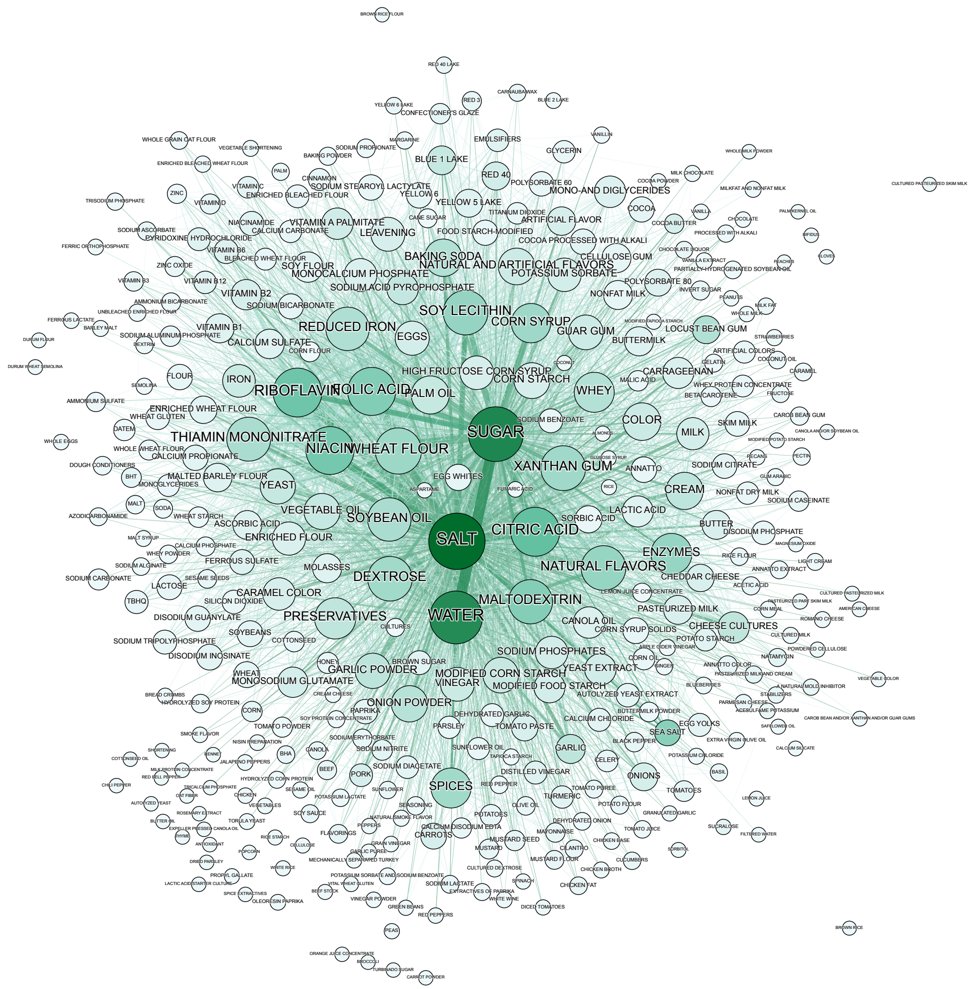
When you buy prepared food (say, a can of baked beans), you buy an elaborate concoction of ingredients: prepared beans, water, sugar, applewood smoked bacon, molasses, textured vegetable protein, and many others. You can think of the ingredients as separate products that happen to be packed together in the can. They occur in the same place at the same time—therefore, they are excellent candidates for becoming product network nodes. By constructing a product network, you can learn which ingredient combinations are most

2. http://www.shmoop.com/othello/antagonist.html

common, whether and how the ingredients group, and which ingredients are central to our food.

You can collect data for a network of ingredients from the website of the United States Department of Agriculture (USDA[3]). There is no need to download all several hundred thousand product descriptions. For starters, we suggest crawling a couple of thousand pages—for example, 925 products with 356 distinct ingredients.

In the following figure, two ingredient nodes are connected if they happen together in more than five food items (the threshold of five was chosen to keep the network connected but not too hairy).

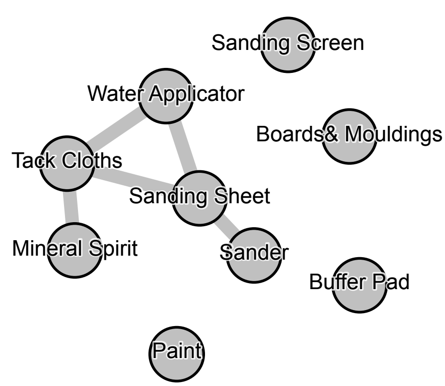


For each node, we calculate its betweenness (color) and eigenvector (size) centralities. The most central nodes represent the core ingredients. Not surprisingly, the top three ingredients are salt, sugar, and water: they go almost into any food item, and they almost always go side by side. The composition of the second ring is less predictable. The next ten most central ingredients are: citric acid (acidifier), maltodextrin (sweetener), xanthan gum (thickener), enzymes (catalysts), natural flavors, wheat flour, niacin (vitamin B), riboflavin (another kind of vitamin B), folic acid (yet another kind of vitamin B), and lecithin (emulsifier). Most of these ingredients are responsible for foundational taste and texture food properties, which explains their position in the network.

Design a Do-It-Yourself Store

Networks of products are common in marketing analysis. Marketing specialists construct product networks to reveal tightly coupled groups of products frequently purchased together. Retailers may compactly stock the products in a group in stores for the ease of shopping. If someone buys a product from a group, they may be reminded to buy other products from the same group. Finally, a group of products may be a stepping stone in a long-term customer project (for example, someone purchasing masonry products may be building a garage and would later need carpentry tools and materials, followed by brushes and paints).

The following figure shows a product network for a pre-painting project, derived from the sales data collected and provided by a Fortune 500 specialty retailer (adapted from *Building Mini-Categories in Product Networks [ZLZ15]*). The network has only nine nodes, four of which are isolated from the other five (we call them isolates—you will learn more about them in *Locate Isolates,* on page 125). Apparently, the isolated products were insignificantly connected to their neighbors, and the network analyst decided to drop the thin edges.



If you were to design an ideal do-it-yourself store, you would put water applicators, tack cloths, sanding sheets, sanders, and mineral spirits on the same shelf, and the other four products on the next shelf. If your customers bought a sanding sheet, but no sander, you (or your recommendation system) would remind them to purchase the sander and another seven items as well.

You learned about two uncommon types of complex networks—semantic networks of words and concepts and co-purchasing-based product networks. The latter can be found in marketing research; the former apply to text analysis and knowledge representation. You will see a complete example of a product network construction and analysis in Chapter 13, *Case Study: Going from Products to Projects*, on page 153. However, before you construct something big, you will learn how to deconstruct a network into compact blocks. In the world of large complex networks, dividing and conquering is the only way to manage complexity. The next chapter will show you how to unearth the network structure.

Divide et impera.

➤ *Attributed to Philip II, King of Macedon*

CHAPTER 11

Unearthing the Network Structure

You're probably not going to be surprised that complex networks have…a complex structure. From the ancient times to modern days, researchers and practitioners have confronted complexity by dividing a complex system into smaller parts—constituents—and then taking a closer look at the parts and making sense out of them. A part could be as small as a single network node or as large as a so-called giant connected component (GCC). (You will meet a GCC later on page 128; for now, it suffices to know that it is giant.) You need a "network-o-scope" to zoom in and out—and you're going to build it in this chapter.

You will learn how to dissect an original complex network into constituents of various sizes, shapes, and types: isolates, connected components, cliques, communities, and k-cores, to mention but a few. You will understand the function and place of each type of constituent in the network analysis workflow. At the end, you will get some suggestions about naming the extracted parts, because, as a rule of thumb, you cannot successfully use something that does not have a name. In other words, you will be ready to *divide and conquer* complex networks. (Which, sadly, did not save the author of these words, King Philip II of Macedon, from an assassination.)

Locate Isolates

The smallest distinct element of any network is an isolate: a node that is not connected to any other node (an isolate can still be connected to itself with a loop edge). Though isolates belong to a bigger network, their very existence is against the networking spirit, because the whole idea of networking is that of connectedness. An example of an isolate in a semantic network is a word that has no synonyms, no homonyms, no antonyms, and no other relationships to any other word (say, "sphygmomanometer" in a network of simple

synonyms—because it has none). An example of an isolate in a product network is an item that nobody ever buys together with any other item. The last meal comes to mind, but then, again, nobody pays for the last meal, so technically it is not even a purchase.

As a network analyst, you want to identify isolates and research the reasons for their isolation. Have you overlooked an edge while constructing the network? (Go over the network construction process one more time.) Have you replaced negative ties in a signed graph with zero-weight positive ties and later discarded them? (Check if there is a better way to preserve negative ties.) Have you sliced a weighted network too aggressively? (*Slice Weighted Networks*, on page 79. Select a lower slicing threshold, if possible.) If the reasons seem to be valid, there is no more need to keep the isolates in the network. Locate them with `nx.isolates(G)`, include their names or count into the final report, and chop the isolates off:

```
G = nx.Graph()
G.add_nodes_from("ABCD") # No edges -- all nodes are isolates
my_isolates = nx.isolates(G)
```

```
❮ ['D', 'C', 'B', 'A']
```

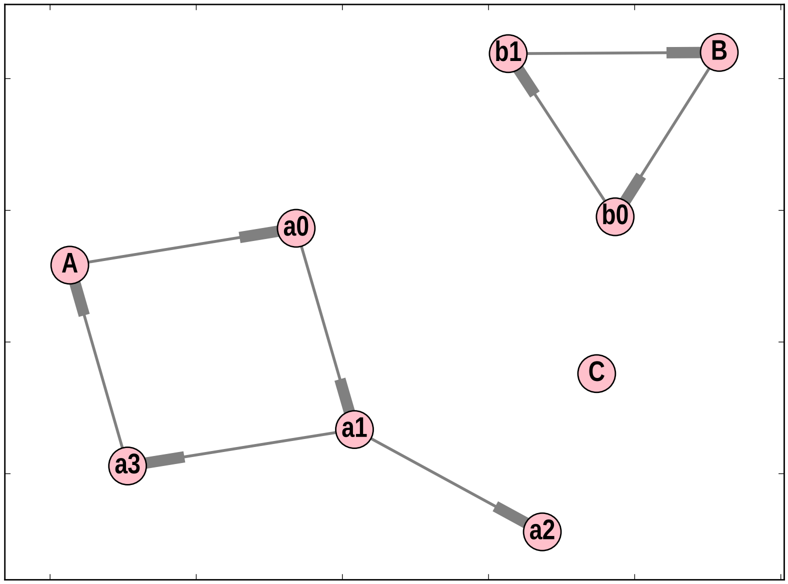
```
G.remove_nodes_from(my_isolates) # No more isolates!
my_isolates = nx.isolates(G)
```

```
❮ []
```

Split Networks into Connected Components

A connected component is a subset of network nodes such that there exists a path (*Think in Terms of Paths*, on page 88) from each node in the subset to any other node in the same subset. An isolate is a special case of a connected component: there is only one node in the subset, so no path is even needed! The figure on page 127 shows a network with three connected components: a larger component A, smaller component B, and isolate C. A fictitious network traveler can get from any node in A to any node in A, but not to any node in B.

If a network is directed, it may have weakly and strongly connected components. In a strongly connected component, there is always a directed path from any node of the component to any other node of the same component. In a weakly connected component, you are allowed to travel one-way streets in the wrong direction (drive responsibly!), if this is what it takes to get from the source to the destination. Both components A and B in the figure are weakly connected, but B is also strongly connected. (No nodes are reachable from *a2*!)



NetworkX provides two families of functions for component analysis. Let's experiment with the network from the previous figure:

make-figures.py
```
F = nx.DiGraph()
F.add_node("C")
F.add_edges_from([("B", "b0"), ("b0", "b1"), ("b1", "B")])
F.add_edges_from([("A", "a0"), ("a0", "a1"), ("a1", "a2"), ("a1", "a3"),
                  ("a3", "A")])
```

Functions nx.connected_components(G) (implemented only for undirected networks), nx.strongly_connected_components(F), and nx.weakly_connected_components(F) (both implemented only for directed networks) take a network of the appropriate type as the parameter and return a generator of sets of nodes that belong to the namesake components. You can coerce the generator to a list, if necessary:

```
list(nx.weakly_connected_components(F))
```

```
[{'A', 'a0', 'a2', 'a1', 'a3'}, {'B', 'b0', 'b1'}, {'C'}]
```

```
list(nx.strongly_connected_components(F))
```

```
[{'a2'}, {'A', 'a0', 'a1', 'a3'}, {'B', 'b0', 'b1'}, {'C'}]
```

```
G = nx.Graph(F)
list(nx.connected_components(G))
```

```
[{'A', 'a0', 'a2', 'a1', 'a3'}, {'B', 'b0', 'b1'}, {'C'}]
```

Note how we convert the directed graph F into an undirected graph G in the last expression. The connected components of the converted graphs are the same as the weakly connected components of the original graph. You can use the obtained node sets to extract the respective subgraphs from the original graph:

```
wcc = nx.subgraph(F, list(nx.weakly_connected_components(F))[1])
len(wcc)
```

```
❮ 2
```

Alternatively, you can use the other family of functions:

- nx.connected_component_subgraphs(G) (implemented only for undirected networks)
- nx.strongly_connected_component_subgraphs(F)
- nx.weakly_connected_component_subgraphs(F) (the latter two functions are implemented only for directed networks)

These functions take a network as the parameter and return a generator of Graph of DiGraph objects, depending on the original network type. So, if you only want to know which nodes belong to what component, the functions without the _subgraphs suffix may save you some time. If you have further operations in mind, go with the second family.

Most co-occurrence networks, by construction, are undirected. Indeed, the fact that items A and B are in the same place at the same time implies that A is in the same place with B and the other way around. That's why for the rest of the chapter, let's assume that all networks are undirected and all connected components are just connected components, without any references to their strength or weakness.

One of the components in a complex network often dominates the others: not so much because it is strong, but because it is giant. The giant connected component (GCC) is simply the largest component by the node count. NetworkX does not provide a function for extracting the GCC, but you can still find it by calling one of the functions mentioned previously, reverse sorting the generated list by size, and taking the first element:

```
comp_gen = nx.connected_components(G)
gcc = sorted(comp_gen, key=len, reverse=True)[0]
```

The size of a GCC in complex networks typically ranges between 80 and 100 percent of the full network size. You may want to treat each smaller component as an indivisible structural unit of the network and focus your attention on the GCC. As a bonus, you will spare yourself from remembering which algorithms and functions apply to disconnected networks and which do not.

What Makes Components Giant?

According to Albert-László Barabási, prominent complex network researcher, most complex networks evolve as a result of preferential attachment. Preferential attachment (also known as the "rich get richer," "80/20," or the Pareto principle) suggests that when a new node joins a network, it is likely to attach itself to a node with the highest degree. Thus, the degree of the node with the highest degree becomes even higher, and the connected component that contains that node grows faster than all other connected components, leading to the emergence of the GCC. The converse is also true: if a network has a GCC, it is likely a result of preferential attachment.

Separate Cores, Shells, Coronas, and Crusts

The only valuable property of a connected component is its connectedness. There is always a way to get from any node A in a component to any other node B in the same component. The property of connectedness is global and, while important for social and communication networks (where paths are responsible for information diffusion), may not be adequate for semantic, product, and other types of networks, where direct or short-haul connections are more essential. Consider a network of synonyms: "emerald" is a synonym of "green," and "green" is a synonym of "ecological," but "ecological" is hardly a synonym of "emerald."

Let's zoom into a connected component (say, in the GCC) and try to find more elements inside.

One of the fundamental tools in modern sociology is core-peripheral analysis. A social network, thereby, consists of two sets of nodes: the core (the nodes that are more or less tightly interconnected) and the periphery (the nodes that are tightly connected to the core, but only weakly, if at all, connected to the other peripheral nodes). The graphs of core-peripheral networks often have a "hairy" appearance: their dense "body" is adorned with "pendulums," multi-edge self-loops, and so on.

Social networks are not the only networks known to have the core-periphery structure. As another example, networks of journal citations reveal the same pattern, where the core consists of papers published in prominent journals in the field, and papers published in marginal journals populate the periphery.

Traditional social network analysis attacks the core-periphery decomposition via fuzzily defined blockmodeling. We, on the other hand, will start by introducing a more mathematically rigid classification of nodes into four categories: cores, shells, coronas, and crusts.

A core or, more accurately, a k-core (where k could be any non-negative integer number) is a subgraph of the original network graph such that each node in the subgraph has at least k neighbors. A 0-core is, naturally, the whole graph. A 1-core is a graph with no isolates. A 2-core is a graph where no node has fewer than two neighbors (no node is a part of a pendulum). Any graph usually has more than one core; the core with the largest possible k is called the main core. A k-core construction process is iterative:

1. Start with the original graph and remove all nodes that have a degree smaller than k and all the incident edges; this will probably result in some of the remaining nodes losing their neighbors and their degree decreasing.

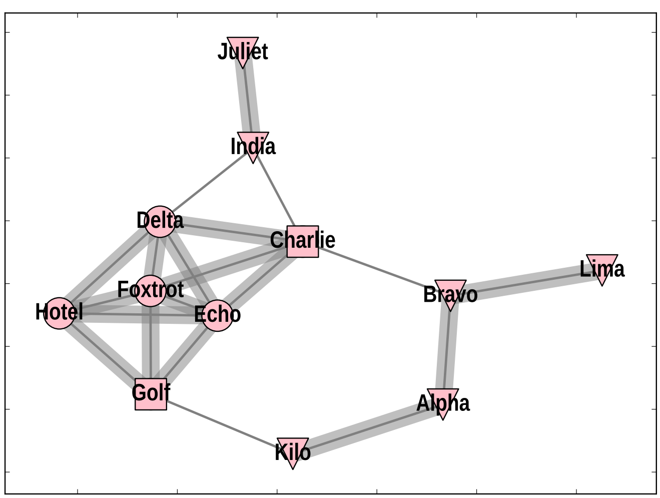
2. Some nodes that have k neighbors or more may have fewer than k neighbors after trimming; remove them, too, and iterate until no remaining node has fewer than k neighbors.

3. The remaining nodes form the k-core.

A k-crust is what is left of the original network when we remove the k-core. In other words, the crust is the periphery.

A core has its internal structure. The subgraph of the k-core in which all nodes have exactly k neighbors in the core is called a k-corona. Unlike crusts, coronas are not necessarily connected and may consist of unconnected components—that is, unconnected within the corona.

Finally, a subset of nodes in k-core but not in (k+1)-core, is called a k-shell. Just like a corona, a shell may consist of components that are not connected within the shell. Let's experiment with the graph from the following figure.



make-figures.py

```
G = nx.Graph(
    (("Alpha", "Bravo"), ("Bravo", "Charlie"), ("Charlie", "Delta"),
     ("Charlie", "Echo"), ("Charlie", "Foxtrot"), ("Delta", "Echo"),
     ("Delta", "Foxtrot"), ("Echo", "Foxtrot"), ("Echo", "Golf"),
     ("Echo", "Hotel"), ("Foxtrot", "Golf"), ("Foxtrot", "Hotel"),
     ("Delta", "Hotel"), ("Golf", "Hotel"), ("Delta", "India"),
     ("Charlie", "India"), ("India", "Juliet"), ("Golf", "Kilo"),
     ("Alpha", "Kilo"), ("Bravo", "Lima")))
```

NetworkX provides a useful collection of functions for calculating all k things. Each of them takes a graph and k as the parameters and returns the namesake core-periphery element (k is optional for nx.k_shell(), nx.k_crust(), and nx.k_core(): they return the main shell, crust, and core by default):

```
nx.k_core(G).nodes() # Round and square nodes and shaded edges
```

```
❮ ['Golf', 'Charlie', 'Delta', 'Hotel', 'Foxtrot', 'Echo']
```

```
nx.k_crust(G).nodes() # Triangular nodes and shaded edges
```

```
❮ ['Lima', 'Bravo', 'Kilo', 'Juliet', 'Alpha', 'India']
```

```
nx.k_shell(G).nodes() # Round and square nodes and shaded edges
```

```
❮ ['Golf', 'Charlie', 'Delta', 'Hotel', 'Foxtrot', 'Echo']
```

```
nx.k_corona(G, k=3).nodes() # Square nodes
```

```
❮ ['Golf', 'Charlie']
```

Extract Cliques

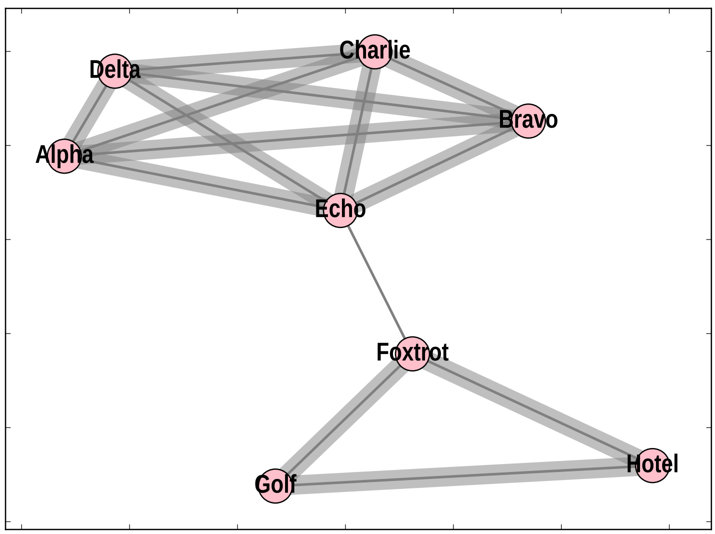
Unlike the smaller components, the GCC and the k-cores are usually too large to be considered a single structural element. Depending on your interpretation of the nodes and edges, you should zoom in even further in a search for smaller network building blocks, such as cliques.

A clique, or, more accurately, a k-clique is a subset of k nodes such that each node is directly connected to each other node in the clique. (We distinguish weak and strong cliques in directed graphs.) Cliques are also known as complete subgraphs. The nodes in a clique may be connected to other nodes as well, but they do not have to—that is, the degree of a node in a k-clique is at least k-1. The principal difference between cliques and connected components is that the path between any two nodes in a clique must have the length of 1, while in a component, the path length is limited only by the graph diameter (*Think in Terms of Paths*, on page 88).

Any single node is a 1-clique, a monad. Any two connected nodes form a 2-clique, a dyad. A triangle of nodes—the result of transitive closure—is a 3-clique, a triad (*Explore Neighborhoods*, on page 84). Monads, dyads, and triads are very common in complex networks.

A maximal clique is a k-clique that cannot be made a (k+1)-clique by adding another node to it. For example, clique (Alpha, Bravo, ..., Echo) in the following figure is a maximal clique, because including any other node (Foxtrot, Golf, or Hotel) into it invalidates the complete connectedness property. For example, if Foxtrot is included, then (Alpha, Bravo, ..., Foxtrot) is not a clique anymore. The largest maximal clique in a network graph is called the maximum clique. (No, I did not invent this terminology!)

Let's experiment with the graph from the following figure:



make-figures.py

```
# Generate a 5-clique
G = nx.complete_graph(5, nx.Graph())
nx.relabel_nodes(G,
        dict(enumerate(("Alpha", "Bravo", "Charlie", "Delta", "Echo"))),
                 copy=False)
# Attach a pigtail to it
G.add_edges_from([
        ("Echo", "Foxtrot"), ("Foxtrot", "Golf"), ("Foxtrot", "Hotel"),
        ("Golf", "Hotel")])
```

NetworkX provides function nx.find_cliques() for finding all maximal cliques in a graph (the largest of which is the maximum clique). The function returns a

list generator, and this time, the use of a generator is not a tribute to Pythonic programming style, but a dire necessity. As a matter of fact, larger cliques, especially maximal and maximum cliques, are rare and hard to find. Finding large cliques is a computationally very hard problem (known as an NP-complete problem) and listing all large cliques requires exponential time. Unfortunately, function `nx.find_cliques()` generates cliques in a random order, but if you want to get only some maximal cliques, not all of them, then you can stop the generator whenever you want. The following code finds all three maximal cliques from the figure (I highlighted the larger two in the figure):

```
list(nx.find_cliques(G))
```

```
[['Golf', 'Hotel', 'Foxtrot'], ['Echo', 'Alpha', 'Bravo',
  'Charlie', 'Delta'], ['Echo', 'Foxtrot']]
```

You have at least two good reasons to search a network for k-cliques: a theoretical and an empirical one. In the theoretical case, you may already have some prior knowledge about the network structure. For example, a marketing specialist may define a project basket in a product network as a collection of products such that they are always purchased together (and, therefore, form a clique when represented as a network). Recognizing k-cliques in a product network almost instantly leads you to the discovery of project baskets. Closely cooperating teams in social and organizational networks are k-cliques, and such are collections of complete synonyms in semantics networks.

In the empirical case, you use cliques as opaque network atoms. If you assume that an edge between two nodes is an indication of their significant similarity, then a complete connectedness within a clique implies overall significant similarity of the member nodes. Thus, you can replace all `k` nodes with one node that represents the entire clique, or with a newly minted "clique-node," potentially significantly simplifying the network topology. Function `nx.make_max_clique_graph()` generates a new graph by replacing each maximal clique with a new synthetic node:

```
synthetic = nx.make_max_clique_graph(G)
synthetic.edges()
```

```
[(1, 3), (2, 3)]
```

Naturally, you can replace cliques with synthetic nodes in the a priori case, too! Just be aware that this function first finds all maximal cliques, with all the NP-completeness implications of finding all maximal cliques.

Recognize Clique Communities

By definition, a clique is a very rigid and sensitive network structure. Removing an edge from a k-clique transforms it into two interwound, partially overlapping, adjacent (k-1)-cliques. In the figure on page 135, subgraphs Alpha–Delta and Alpha–Charlie, Echo are 4-cliques each, but the whole network is not a 5-clique, because the edge Delta–Echo is missing (I show it as a dashed line).

Intuitively, you may feel that all k nodes somehow belong together and the missing edge could have been a victim of a measurement or data entry error or improper slicing (*Slice Weighted Networks*, on page 79) or conversion from a directed graph. Nonetheless, nx.find_cliques() will not recognize your k nodes as a clique (because they are not). Instead, it will report two smaller cliques, leaving it up to you to notice that they actually share k-1 nodes and their separation may have been caused by a missing edge.

Luckily, NetworkX supports k-clique communities. A k-clique community is a union of all k-cliques that can be reached through adjacent k-cliques. The process of reaching all cliques in the union is called *clique percolation [PDFV05]*.

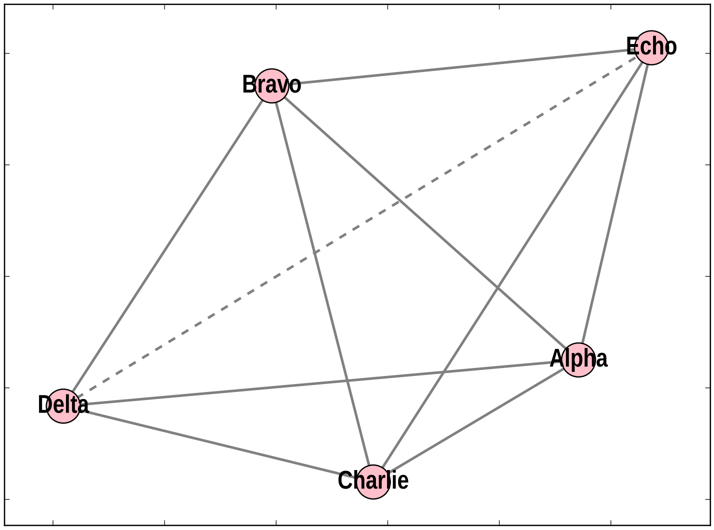
When a Community Is Not a Community and a Cluster Is Not a Cluster

Anthropologists and social scientists have a different idea of a community and may get easily confused at this point. For them, a community is a tightly knit group of people, not a group of abstract nodes. To avoid getting into pointless terminological fights, I will sometimes refer to communities as clusters. Unfortunately, data scientists who may be reading this book, too, have a different definition of a cluster. Apparently, when it comes to network analysis, terminological fights are unavoidable.

K-clique communities in complex networks are a curse and a blessing. Why are clique communities a blessing?

They provide a flexible substitute for inflexible proper cliques. True, the nodes in a community are not in general directly interconnected; however, if the relationship represented by the edges is actually transitive (if A is adjacent to B, and B is adjacent to C, then A is supposed to be adjacent to C—as it would be in the case of two products always purchased together) and the missing edges result from network construction imperfection, then the length of the path between any two nodes in a clique community does not really matter.

Why are they a curse, then?



Just like strict cliques, clique communities do not necessarily partition the network and can overlap with other clique communities. In other words, the same node may belong to more than one clique or clique community. This may or may not be what you want.

What's worse, if the relationship represented by the edges is only approximately transient (if A is adjacent to B, and B is adjacent to C, then A is not necessarily adjacent to C—as it would be in the case of personal friendship), then two nodes separated by a multi-edge path may actually have little or nothing shared, and their membership in the same clique community would be questionable.

Only you can determine whether clique communities are appropriate for your network. But once you do, here's the function for doing the dirty job:

```
list(nx.k_clique_communities(G, k=3))
```

```
❮ [frozenset({'Golf', 'Hotel', 'Foxtrot'}),
   frozenset({'Alpha', 'Charlie', 'Bravo', 'Echo', 'Delta'})]
```

Note that a k-clique community always has at least `k` nodes!

Frozen Sets

A frozenset is an immutable version of a Python set. Because of its immutability, it can be used as key in a dictionary, but it can be cast to a set, if any modifications are necessary.

Outline Modularity-Based Communities

The fuzziest and most flexible form of node organization in a complex network is network communities based on modularity. They are sometimes also called clusters or groups, and are not to be confused with clique communities (*Recognize Clique Communities, on page 134*).

This section uses community.

Let's start with modularity first, and assume that the network has been already partitioned into non-overlapping communities (later you'll figure out how). According to *Newman's definition [New06]*, modularity `m` is the fraction of the edges that fall within the given communities minus the expected fraction if edges were distributed at random, while conserving the nodes degrees. The value of `m` is in the range from -0.5 (inclusive) to 1 (exclusive). If most of the edges are incident to the nodes within the same community, the modularity is very high, close (but not equal) to 1, and the proposed partition describes a very good community structure. The modularity of -0.5 means that the nodes within the same community are not adjacent at all—the proposed community structure is worse than random; in fact, you are probably dealing with anti-communities that induce bi- and multi-partite networks (as in *Project Bipartite Networks, on page 178*).

Ideally, you want to partition a network in such a way that the modularity is as high as possible. The modularity of 0.6 and above corresponds to networks that have a clearly visible community structure. Unfortunately, getting the largest modularity is hard for at least three reasons:

- The problem of optimal partitioning is NP-complete with respect to the network size. To find the best partition, you should calculate modularity for every possible partition and then select the best one. The number of partitions is simply too large, and the problem does not have a feasible exact solution for any non-trivial graph.
- Approximate solutions (for example, the *most popular Louvain algorithm [BGLL08]*) do a pretty good job, but some of them are probabilistic, which means every time you run them, you may end up having slightly different partitions.
- The resolution of modularity-based methods scales poorly, and they overlook small communities in large networks. A plausible solution is to partition the network recursively into smaller and smaller communities.

Anaconda, the most popular Python distribution, does not currently include tools for modularity-based community detection. Fortunately, the tool exists

and can be easily installed via pip. It is called python-louvain. The externally visible name of the module is community, and you import it under this name.

Module community uses the louvain algorithm that optimizes network modularity. The discovered communities are represented as a partition: a dictionary with node labels as keys and integer community identifiers as values. The module also calculates the modularity of the partition with respect to the original network.

```
part = community.best_partition(G)
```

```
{'Golf': 0, 'Bravo': 1, 'Delta': 1, 'Hotel': 0, 'Foxtrot': 0,
 'Charlie': 1, 'Alpha': 1, 'Echo': 1}
```

```
community.modularity(part, G)
```

```
0.3035714285714286
```

A Faster Way

There is a faster implementation of the louvain algorithm provided through the module louvain (must be installed separately, too). However, it works only with graphs constructed in iGraph, not in NetworkX.

Just as with cliques and clique communities, you may want to replace the smaller structural elements of a large network with synthetic nodes and build an induced graph:

```
induced = community.induced_graph(part, G)
induced.nodes()
```

```
[0, 1]
```

```
induced.edges()
```

```
[(0, 0, {'weight': 10}), (0, 1, {'weight': 1}), (1, 1, {'weight': 3})]
```

Note that the induced graph is weighted and has loops. The weight of an induced edge incident to the synthetic community nodes is the number of edges in the original network that are incident to the nodes in the respective communities.

Explore Modularity-Based Communities with Pandas

If you're familiar with Pandas, you can convert a network partition part (a dictionary) into a Series for the ease of further processing. You can see the total number of communities, their sizes, and which nodes belong to which community:

```
part_as_series = pd.Series(part)
part_as_series.sort_values()
```

```
Foxtrot    0
Golf       0
Hotel      0
Alpha      1
Bravo      1
Charlie    1
Delta      1
Echo       1
dtype: int64
```

How big are the communities?

```
part_as_series.value_counts()
```

```
0    5
1    3
dtype: int64
```

Perform Blockmodeling

The construction of the graph of maximal cliques or communities is a special case of blockmodeling—grouping network nodes according to some meaningful definition of equivalence and replacing them with synthetic "supernodes." A more general function nx.blockmodel(G,part) takes a graph G and its partition part as a list of node collections (lists or sets), and creates an induced graph. Unlike nx.make_max_clique_graph() and community.induced_graph(), nx.blockmodel() requires the partition includes every node in the original graph exactly once. You can manually remove the offending overlapping clique from a clique partition, if you want:

```
cliques = list(nx.find_cliques(G))
```

```
[['Golf', 'Hotel', 'Foxtrot'], ['Echo', 'Alpha', 'Bravo',
  'Charlie', 'Delta'], ['Echo', 'Foxtrot']]---not good!
```

```
synthetic = nx.blockmodel(G, [cliques[0], cliques[1]])
synthetic.edges()
```

```
[(0, 1)]
```

Not All Blockmodeling Leads to the Same Rome

For social scientists, "blockmodeling" often means a very different thing: separating a network into the core and periphery by way of rearranging rows and columns of the incidence matrix. Blockmodeling as understood by complex network analysts is a generalization of the core-periphery decomposition.

Name Extracted Blocks

From the data scientific point of view, network analysis at the macroscopic level (extraction of communities, cliques, and other structural blocks) is an example of unsupervised machine learning. The goal of unsupervised machine learning is to infer a network's hidden structure in the absence of "labels": node and edge attributes (except, perhaps, the edge weights).

The unearthed blocks suffer from two major interrelated problems:

- It is not clear what they mean.
- They are nameless.

In fact, if you knew the purpose or nature of a block, you would give it a name, and if you knew the name, you would guess what its purpose or nature is.

Selecting a good name for a block can be done in at least three ways.

- You can use your intelligence: look at the individual node labels and generalize. A block that has labels "car," "truck," "train," and "sled" probably deserves to be called "land transportation," and "hand," "arm," "leg," "head," and "chest" belong to the block "body parts." If unsure or confused, hire a subject matter expert (SME) whose job is to know why nodes X and Y ended up in the same block.

- Better yet, hire a lot of subject-matter experts—or sort-of-experts. *Amazon Mechanical Turk (AMT) [BKG11]* offers a way to put any question to literally thousands of people ("workers," in the AMT terminology) for a very modest price. Ask 10,000 AMT workers what "foos," "bars," and "foobars" have in common. If the terms have anything in common at all, you will most probably get an answer supported by a solid majority of workers.

- Finally, if you cannot hire an SME and would rather not mess with AMT, you can still generate block labels from its data. If the nodes in a block differ—for example, in size or weight—take the largest of them (say, "head") and use its label to synthesize the block label (for example, "the 'head' group"; see *Interpret the Results*, on page 150 for a better example). If all nodes have the same attributes or have no attributes at all, choose the first node in the alphabetic order ("the 'arm' group").

In this chapter, you learned how to dissect a complex network and explore its anatomy. You know how to locate isolated nodes and components; identify cores, shells, coronas, and crusts; and compute node cliques, clique communities, and modularity-based communities (sometimes referred to as clusters). Now, it is time to apply the freshly minted "network-o-scope" to a couple of real-world semantic and product networks.

...Culture opens the sense of beauty.

> *Ralph Waldo Emerson, American essayist, lecturer, and poet*

CHAPTER 12

Case Study: Performing Cultural Domain Analysis

This chapter uses NLTK, Pandas, NumPy.

Cultural domain analysis (CDA, *Analyzing Qualitative Data. Systematic Approaches [BWR17]*) is the study of how people in groups think about lists of terms that somehow go together and how this thinking differs between groups. Some people associate "candle" with "Christmas," others with "hurricane" and "blackout," and yet others with a "self-injury" (cutting/burning their skin) toolset. Anthropologists, ethnographers, psychologists, and sociologists use CDA to understand semantic mindscapes of social, ethnic, religious, professional, and other groups. Before personal computers and specialized CDA software became available, social scientists used to do CDA essentially by hand. But not anymore! Python comes to rescue.

You don't have to be an anthropologist, ethnographer, psychologist, or sociologist to read this chapter. Regardless of your background, you will learn how to harvest semantic data from a popular blogging website and cache it locally for further efficient access. You will see how to convert natural language units into terms and build, analyze, and interpret a semantic network reflecting the interests of the fans of *The Good Wife*, a CBS TV show. Hopefully, you will be able to extend the same approach to other shows, other websites, and other tag corpora.

The complete code for this case study is available in the file lj.py.[1]

1. pragprog.com/titles/dzcnapy/source_code

Get the Terms

Start by importing all necessary modules and defining the domain (LJ community) of interest. We suggest using Pandas and NumPy, the power tools of data science, and NLTK—the Natural Language Toolkit—in this project, as well as some other libraries, so you need to import them. (If you last used them a while ago, you might want to *blow the dust off your skill set [Zin16]*.)

lj.py
```
import urllib.request, os.path, pickle # Download and cache
import nltk # Convert text to terms
import networkx as nx, community # Build and analyze the network
import pandas as pd, numpy as np # Data science power tools
```

Your next step is to get and cache term lists. A term is a unit of CDA. It can be a word, a word group, a word stem, or even an emoticon (emoji). CDA looks into similarities between terms that are shared among a reasonably homogeneous group of people. So, you need to find a reasonably homogeneous group of individuals, a list of terms, and a way to assess their similarity.

A great source of semantic data is LiveJournal (LJ)—a collection of individual and communal blogs with elements of a massive online social network.[2] LiveJournal has an open, easily accessible API, and encourages the use of public data for research. Unfortunately, LJ membership and activity peaked in the early 2000s, but the site still hosts some lively blogging communities (such as the celebrity gossip blog "Oh No They Didn't!" [3]), and it serves rich layers of historical data.

LJ communities consist of individual members that have their private blogs, profiles, friend lists, and interests (online identity markers). In fact, LJ treats communities and users as same-class citizens: communities, like individuals, have their profiles, interests, and even "friends." The URLs of user/community profiles, interest lists, and friend lists have a regular structure. If *thegoodwife_cbs* is the name of a community (you will use it in the rest of the study), then thegoodwife-cbs.livejournal.com/profile/ is the URL of the community profile (note that the underscore was replaced by a dash), www.livejournal.com/misc/fdata.bml?user=thegoodwife_cbs&comm=1 is the friend list, and www.livejournal.com/misc/interestdata.bml?user=thegoodwife_cbs is the interest list.

For the purpose of this mini-study, let's define two terms to be similar from the perspective of an LJ community if they are consistently listed together on

2. www.livejournal.com
3. ohnotheydidnt.livejournal.com

different interest lists of the community members. Your first job is to obtain and process the community membership list. A typical list looks like this:

```
# Note: Polite data miners cache on their end. Impolite ones get banned.
# Note: thegoodwife_cbs is a community account
P> exploding
P> poocat
«…more members…»
P< brooketiffany
P< harperjohnson
```

The first line of the document makes a significant point: if you download something once, you should not download it again. Create the directory cache and store all downloaded data into it. If you run CDA on the same data again, it will hopefully still be in the cache, assuming that interest lists and community membership are reasonably stable.

lj.py

```
LJ_BASE = "http://www.livejournal.com/misc"
DOMAIN_NAME = "thegoodwife_cbs"

cache_d = "cache/" + DOMAIN_NAME + ".pickle"
if not os.path.isfile(cache_d):
    domain = download(LJ_BASE, DOMAIN_NAME)
    if not path.os.isdir("cache"):
        os.mkdir("cache")
    with open(cache_d, "wb") as ofile:
        pickle.dump(domain, ofile)
else:
    with open(cache_d, "rb") as ifile:
        domain = pickle.load(ifile)
```

This code fragment uses the module pickle—native Python data serializer (*Export and Import Networks*, on page 30). If the cache directory and file exist, function pickle.load() deserializes the data object. Otherwise, create the directory and file and call function pickle.dump() to serialize the data object domain and save it into the file. Note that you must open the file in the binary mode. The compressed cached pickle file is available as thegoodwife_cbs.pickle.zip.[4]

The rest of the community membership list on page 143 is a two-column table. The first column represents some subtle aspects of membership types ("P" for individual users, "C" for communities, "<" for "friends," ">" for "friends-of"); the second column has usernames. To keep your code modular, write a function download(domain_name) that takes care of this and similar tables.

4. pragprog.com/titles/dzcnapy/source_code

lj.py

```
  def download(base, domain_name):
      """
      Download interest data from the domain_name community on
      LiveJournal, convert interests to tags, create a domain DataFrame
      """
1     members_url = "{}/fdata.bml?user={}&comm=1".format(base, domain_name)
      members = pd.read_table(members_url, sep=" ",
                              comment="#", names=("direction", "uid"))

2     wnl = nltk.WordNetLemmatizer()
      stop = set(nltk.corpus.stopwords.words('english')) | set(('&'))
3     term_vectors = []
4     for user in members.uid.unique():
          print("Loading {}".format(user)) # Progress indicator
          user_url = "{}/interestdata.bml?user={}".format(base, user)

5         try:
              with urllib.request.urlopen(user_url) as source:
                  raw_interests = [line.decode().lower().strip()
                                   for line in source.readlines()]
          except:
              print("Could not open {}".format(user_url)) # Error message
              continue

          if raw_interests[0] == '! invalid user, or no interests':
              continue

6         interests = [" ".join(wnl.lemmatize(w)
                                for w in nltk.wordpunct_tokenize(line)[2:]
                                if w not in stop)
                       for line in raw_interests
                       if line and line[0] != "#"]

7         interests = set(interest for interest in interests if interest)
8         term_vectors.append(pd.Series(index=interests, name=user).fillna(1))

      return pd.DataFrame().join(term_vectors, how="outer").fillna(0)\
9         .astype(int)
```

1. Convert the interest table into a two-column DataFrame.

2. Prepare a WordNet lemmatizer wnl—a tool for converting words into lemmas —and a list of stopwords stop, extended to include “&.” You will need this list to eliminate too frequent words. Coerce the standard list of stopwords into a set for faster lookup, because the list lookups in Python are notoriously slow.

Your next step is to download all interest lists, convert interests into terms (they are not always the same!), and combine all term lists into a term matrix —a matrix whose columns are term lists. An interest list looks very similar to the membership list—even the call to caching is the same:

```
# Note: Polite data miners cache on their end. Impolite ones get banned.
# <intid> <intcount> <interest …>
18576742 1 +5 sexterity
624552 7 a beautiful mess
18576716 1 any more hot chicks?
44870 28 seriously?
1638195 94 shiny!
«…more interests…»
```

This is still a table (showing interest ID, the system-wide rank of the interest, and the actual interest), but the number of columns differs, depending on how many words are in the interest description. Pandas DataFrames are poor parsers for irregular texts; tackle the columns by hand, using low-level Python tools. Note that if the username is not found or the user has not declared any interests, the content is entirely different:

```
! invalid user, or no interests
```

So, let's continue the function code inspection:

❸ Set up an empty list accumulator term_vectors. At the end of the loop, it becomes a list of term vectors—raw material for the term matrix.

❹ Loop through all unique usernames in the community, because you need the URLs of their interest lists.

❺ Obtain an interest list raw_interests as a Python list of strings for each distinct community member in the try-except block. If the URL fails to open (for any reason beyond your control), the script does not crash but politely informs the programmer of the failure and proceeds to the next user. The same thing happens when the user has no interests or does not exist at all. If everything goes fine, decode and strip each string of trailing whitespaces. LJ interest lists are always in lowercase, but if they are not, a call to lower() ensures that in the rest of the script you compare apples to apples.

❻ Split each non-empty, non-commented interest into individual words with nltk.wordpunct_tokenize(). There may be more than one word in an interest description, and some or all of these words may be forms of other words (as in "chicks"—"chick"). Text analysis practitioners preach different ways of handling word forms. Some suggest to leave the forms alone and treat them as words on their own. Some advocate lemmatizing or stemming: reducing a form to its lemma (the standard representation of the word) or to the stem (the smallest meaningful part of the word to which affixes can be attached). A lemmatizer reduces "programmers" to one "programmer," and a stemmer, depending on its zeal, yields a "programm" or even a "program." Let's follow the lemmatizing crowd. Furthermore, almost

everyone agrees that certain words (such as "a," "the," and "and") should never be counted at all. Remove them.

7. As a result of lemmatization, stemming, and stopword elimination, a term list may end up having duplicates (for example, "the chicks" and "a chick" may both become "chick"). Convert each term list into a set. Surely, sets have no duplicates.

8. Your goal is to produce a term vector model (TVM)—a table where rows are terms and columns are community members.[5] In the Pandas language, this table is known as DataFrame, and its columns are known as Series. Transform a term list to a Series, where the individual terms become the Series index, and all values are set to 1s, like this:

```
print(term_vectors)
```

```
shiny!           1
+5 sexterity     1
big damn hero    1
«…more terms…»
Name: twentyplanes, dtype: float64
```

9. Finally, join all Series into a DataFrame. This operation involves the mystery of data alignment, whereby all participating Series are stretched vertically to have their row labels (terms!) aligned. Such stretching almost inevitably produces empty cells in the frame. Fill them up with zeros: an empty cell in row A and column B signals that the term A was not on the list B. The resulting variable DataFrame has 12,437 rows and adequately represents the cultural domain of LiveJournal users interested in the CBS show *The Good Wife*.

Build the Term Network

The next CDA step requires that you build a network of terms: a graph where nodes represent terms, and (weighted) edges represent their similarities.

You could have included all 12,437 discovered terms in the graph, but some of them are mentioned only once or twice (which is expected, given Zipf's law[6]). Rather than wondering why the less frequently used words are in fact less frequently used, remove all rows with fewer than ten occurrences, but provide an option of changing the cut-off value MIN_SUPPORT in the future. At this point, you might wish that Python had first-class constants, but

5. en.wikipedia.org/wiki/Vector_space_model
6. mathworld.wolfram.com/ZipfsLaw.html

nonetheless spell MIN_SUPPORT in all capital letters. DataFrame limited is a truncated version of domain: it has only 319 rows.

Zipf's Law

Zipf's law states that given some corpus of terms drawn from a natural language, the frequency of any word is inversely proportional to its rank by frequency. In other words, if the frequency of the most frequent term is f_0, then the frequency of the next most frequent term is $f_0/2$, and so on, and the frequency of the nth term is $f_0/n\alpha$. The same law (with the slightly different exponent α) applies to population ranks of cities, corporation sizes, income rankings, and more. The continuous form of the discrete Zipf law is known as Pareto distribution, and Zipf's law is a special case of the power law (mentioned on page 106).

lj.py
```
MIN_SUPPORT = 10
sums = domain.sum(axis=1)
limited = domain[sums >= MIN_SUPPORT]
```

Since you want to build a network based on co-occurrences, you can consider two terms as similar if different community members frequently use them together. Calculate the matrix of co-occurrence by matrix-multiplying the limited DataFrame by itself.

The resulting square DataFrame cooc contains the total counts of all terms on the main diagonal (suppress them by multiplying the matrix by an inverted identity matrix) and the counts of co-occurrences elsewhere (they will eventually become weighted network edges).

lj.py
```
cooc = limited.dot(limited.T) * (1 - np.eye(limited.shape[0]))
```

Slice the Network

Now, you must make another painful decision: which matrix elements become edges and which get discarded? *Slice Weighted Networks, on page 79*, explains the slicing philosophy. Choose the slicing threshold, SLICING, to be six. Higher SLICING results in many small communities. Lower SLICING results in few large communities. Six seems to be a good compromise between count and size.

The resulting matrix is very sparse (every cell represent an edge, but we agreed to have as few edges as possible!). Stack and normalize it—essentially convert into a sparse matrix, where each row represents a significant edge and its weight. Since NetworkX prefers to deal with Python (rather than Pandas) data structures, convert the weights to a dictionary:

lj.py
```
SLICING = 6
weights = cooc[cooc >= SLICING]
weights = weights.stack()
weights = weights / weights.max()
cd_network = weights.to_dict()
cd_network = {key:float(value) for key,value in cd_network.items()}
```

You are just one step away from having an amazingly structured network. Let's create a new empty graph, populate it with the edges from the dictionary, and update the "weight" edge attributes:

lj.py
```
tag_network = nx.Graph()
tag_network.add_edges_from(cd_network)
nx.set_edge_attributes(tag_network, "weight", cd_network)
```

The constructed network with the added attributes is your first fascinating result; save it without hesitation into a GraphML file in a specially created directory results (*Share and Preserve Networks*, on page 29).

lj.py
```
if not os.path.isdir("results"):
    os.mkdir("results")

with open("results/" + DOMAIN_NAME + ".graphml", "wb") as ofile:
    nx.write_graphml(tag_network, ofile)
```

If you had no access to non-Anaconda modules, you would abandon Python at this point and switch to interactive software like Pajek,[7] UCINET[8] (which are outside the scope of this book), or Gephi (Chapter 4, *Introducing Gephi*, on page 31) for further network analysis. Fortunately, Python has the community module (*Outline Modularity-Based Communities*, on page 136) that will let you stay in the same program for the entire analysis cycle.

Extract and Name Term Communities

The modularity of the new network is quite poor (we suggested on page 136 that a network is definitely modular only when the modularity is 0.6 or above):

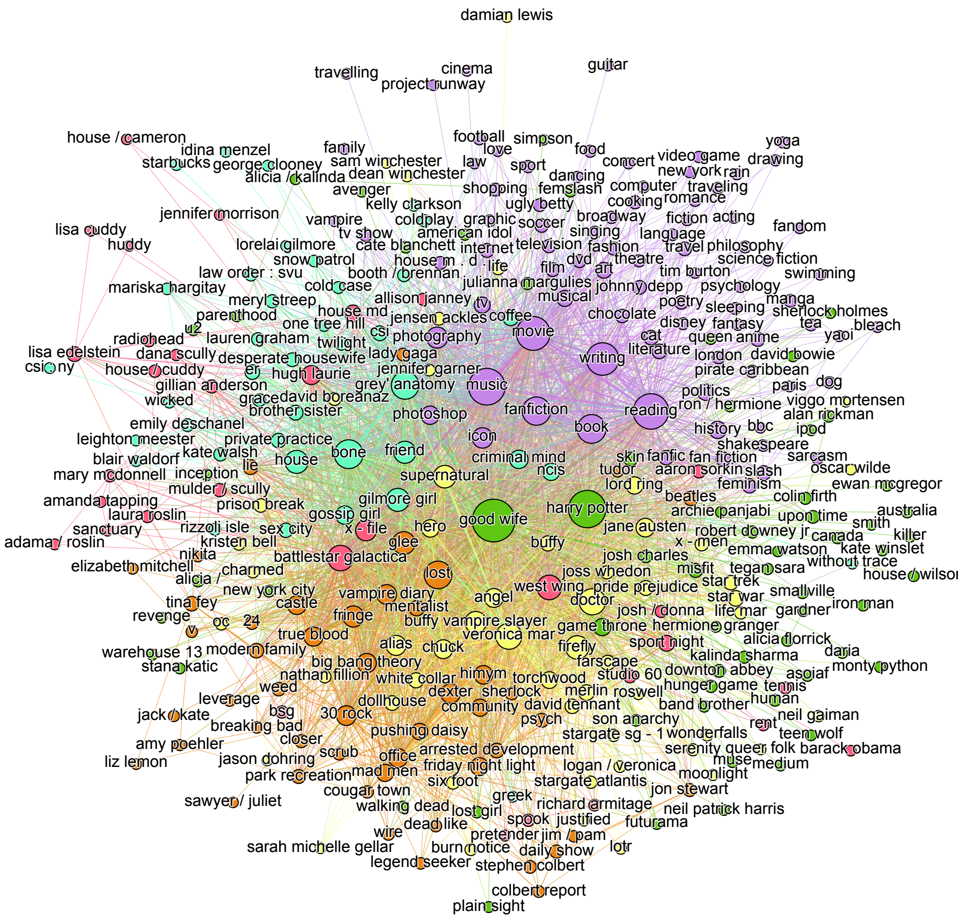
lj.py
```
partition = community.best_partition(tag_network)
print("Modularity: {}".format(community.modularity(partition,
                                                   tag_network)))
nx.set_node_attributes(tag_network, "part", partition)
```

```
❮ Modularity: 0.15815567681142356
```

7. vlado.fmf.uni-lj.si/pub/networks/pajek/
8. sites.google.com/site/ucinetsoftware/home

Apparently, counting raw co-occurrences is not the best way to describe similarities—indeed, correlation-based networks are much more flexible, and you will learn about them later (Chapter 14, *Similarity-Based Networks*, on page 163). However, even the coarse network that you have allows some meaningful interpretation.

The partition that you extracted precisely defines the term communities. (Add them as an attribute *part* to the network nodes.) The following figure shows the whole network of tags. The node diameter represents the number of times the corresponding tag was mentioned in the corpus, and the node colors match the term communities.



So, you extracted the communities, but they are nameless. You could have explored them with your bare eyes and come up with some proper names, but then you would not be a hardcore Python programmer after that, would you? Instead, add a cherry on the cake and write another seven lines of code

that locate the five most frequently used terms per cluster. Hopefully, they indeed describe the content.

You need a little helper function describe_cluster(terms_df). The function takes a DataFrame of terms in one community, extracts the namesake rows from the original domain, calculates their use frequency, and returns the top HOW_MANY performers.

lj.py
```
HOW_MANY = 5
def describe_cluster(terms_df):
    # terms_df is a DataFrame; select the matching rows from "domain"
    rows = domain.join(terms_df, how="inner")
    # Calculate row sums, sort them, get the last HOW_MANY
    top_N = rows.sum(axis=1).sort_values(ascending=False)[:HOW_MANY]
    # What labels do they have?
    return top_N.index.values
```

Finally, convert the partition into a DataFrame, group the rows by their partition ID, and beg the helper to come up with a name for each community.

lj.py
```
tag_clusters = pd.DataFrame({"part_id" : pd.Series(partition)})
results = tag_clusters.groupby("part_id").apply(describe_cluster)
for r in results:
  print("-- {}".format("; ".join(r.tolist())))
```

Surprisingly, it works!

```
-- good wife; harry potter; game throne; misfit; skin
-- music; reading; movie; writing; book
-- bone; grey ' anatomy; gilmore girl; house; friend
-- battlestar galactica; west wing; x - file; hugh laurie; house md
-- lost; glee; fringe; vampire diary; 30 rock
-- doctor; veronica mar; firefly; supernatural; buffy vampire slayer
```

Each line shows up to five most frequently mentioned terms per cluster (the terms are separated by semicolons). Some terms look strange and barely recognizable ("x - file")—but remember all those rigorous transformations that they had to go through, such as lemmatizing! Some terms are duplicates ("house md"—"house"), but this simply means that the transformations were not rigorous enough.

Interpret the Results

There are two levels of interpretation of the CDA results.

At the lower level, you can conclude that all 319 terms selected for analysis are important for *The Good Wife* viewers—otherwise, the viewers would not

have selected them. Moreover, some of the terms go together more often than the others. You can see that "bone[s]," "grey '[s] anatomy," and "gilmore girl[s]" have something to do with each other (hint: Sean Gunn starred in all three series); "veronica mar[s]" and "firefly" are TV shows that were both prematurely canceled; "music," "reading," and "writing" are popular activities... You can make these mechanistic conclusions without having the slightest idea about the cultural domain.

At the higher level, you might be an ethnographer, anthropologist, psychologist, or sociologist interested in the mindscape of *The Good Wife* fans. In particular, you might want to compare their mindscape to the mindscapes of, say, *Harry Potter* or *House, M.D.* fans. However, since you are a humble computer programmer or data scientist, you shall entrust the higher level interpretation to the SMEs.

This case study guided you from selecting an online blogging (or, rather, gossiping) community to constructing and partially interpreting a cultural domain—a semantic network of terms partitioned into six term collections. The method is fairly extensible and can be applied to other topical communities.

You have seen a network of words, but you have not seen it all. In the next chapter, we will show you a network of cosmetics!

In Constantinople there are some persons, particularly Armenians, who devote themselves to the preparation of cosmetics, and obtain large sums of money from those desirous of learning this art.

➤ *G. W. Septimus Piesse, British perfumer*

CHAPTER 13

Case Study: Going from Products to Projects

This chapter uses Matplotlib, community.

One of the goals of product network analysis is to identify nontrivial collections of co-purchased or co-recommended products. We can treat such collections as "customer projects" or "toolsets." You can find these networks of products frequently purchased together or recommended to be used together in marketing, advertising, and similar business disciplines.

As an example of product network analysis, let's have a look at cosmetics sold by Sephora®. In this case study, you will learn how to convert a CSV data file with cosmetics co-purchasing data into a complex network with the help of csv, itertools, and collections libraries. You will calculate attribute assortativity of the complex network and blockmodel it—construct its higher-level representation as an induced graph. Finally, you will use graphviz_layout() to produce a picture of the network without invoking any non-Python software.

Read Data

Most products on Sephora's website have *"Use With"* recommendations: one or more other products that the Sephora staff recommends customers purchase in conjunction with the original product.[1] For each product, the website contains plenty of characterizing information, such as brand, category, price, volume, and star rating. Each product is uniquely identified by the store SKU number and an alphanumeric ID. We will build a network of *"Use With"*

1. www.sephora.com

products created from previously acquired data and explore its structure concerning the product categories.

Crawling Sephora

If you want to download information about a specific product, you can program a crawling procedure using modules urllib.request for the actual download, and BeautifulSoup (bs4) for parsing the HTML response. (Both modules are outside of the scope of this book.) Sephora's website has a straightforward organization. Once you extract the IDs of the recommended products, you can repeat the download/parse cycle until your script finds no more new products. Since the *"Use With"* network is disconnected, you will need to run the crawling procedure more than once, starting from randomly selected products, to harvest all or at least the largest components.

For your convenience, we provide the raw data for the network construction in two CSV files. File use-with.csv has 3,943 rows: the first row is the header; the remaining rows contain network edges as edge ID (not needed in this case study), start product node, and end product node. We assume that the network is undirected (in reality it is not). File product.csv has 2,976 rows: the first row is the header; the remaining rows describe product nodes (one node per row) and contain product attributes as product ID, brand, star rating, and category. The latter two attributes may be empty.

As always, we start by importing all the necessary modules (the Aside titled "Where to Import?" on page 102 explains why):

products.py
```
import csv
from collections import Counter
from operator import itemgetter
from itertools import chain, groupby
import networkx as nx
from networkx.drawing.nx_agraph import graphviz_layout
import community
import matplotlib.pyplot as plt
import dzcnapy_plotlib as dzcnapy
```

Our next step is to read the edges and product attributes from the files, construct a vanilla network, and decorate its nodes with attributes (*Add Attributes*, on page 23):

products.py
```
with open("use-with.csv") as usewith_file:
    reader = csv.reader(usewith_file)
➤   next(reader)
    G = nx.from_edgelist((n1, n2) for _, n1, n2 in reader)
```

```
with open("products.csv") as product_file:
    reader = csv.reader(product_file)
    next(reader)

    brands = {}
    cats = {}
    star_ratings = {}

    for ppid, brand, star_rating, category in reader:
        brands[ppid] = brand
        cats[ppid] = category
        star_ratings[ppid] = float(star_rating if star_rating else 0)

# Set node attributes, based on product attributes
attributes = {"brand" : brands, "category" : cats, "star" : star_ratings}
for att_name, att_value in attributes.items():
    nx.set_node_attributes(G, att_name, att_value)
```

Note how we use next(reader) to skip the header rows in the first two highlighted lines, and how we impute zero star rating for the rows that do not have the star rating field in the last highlighted line.

Analyze the Networks

The resulting graph G has 2,975 nodes and 3,162 edges. It is very sparse:

```
print(nx.density(G))
```

```
0.0007147660678259198
```

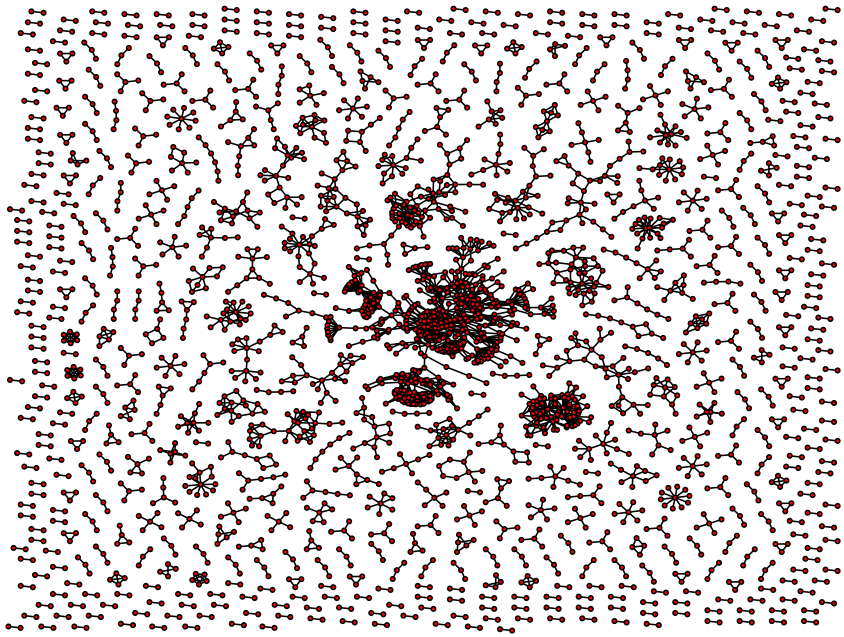
It also has a lot of small connected components with two to four nodes, as shown in the figure on page 156. (You can call nx.connected_components(G) and measure the size and count of them on your own.)

To keep the case simple, we consider only the largest component (the GCC). We sort all components of G by size, select the last one (the largest!), join the respective label lists into one with chain.from_iterable(), and extract the subgraph induced by these nodes. We store the resulting subgraph in the variable called gccs:

products.py

```
TOP_HOWMANY = 1
gccs_nodes = chain.from_iterable(sorted(nx.connected_components(G),
                                        key=len)[-TOP_HOWMANY:])
gccs = nx.subgraph(G, gccs_nodes)
```

The subgraph contains 25 percent of nodes and 36 percent of edges from the original graph, and it also has the most interesting structural elements. If you don’t fancy these numbers, simply change TOP_HOWMANY.



So, what can we say about the distribution of the graph attributes? Do neighbors tend to be assortative or disassortative (*Estimate Network Uniformity Through Assortativity*, on page 97)? Let's find out:

products.py

```
for att_name in attributes:
    print("Assortativity by {}: {}"\
              .format(att_name,
                      nx.attribute_assortativity_coefficient(gccs, att_name)))
```

```
Assortativity by category: 0.03577904976206569
Assortativity by brand: 0.8687551723142831
Assortativity by star: -0.0058012311220827645
```

The news is mixed. On the one hand, connected products are very likely sold under the same brand—because cosmetic brands provide comprehensive toolkits! On the other hand, connected products belong to different categories —indeed, why would one buy two tools from the same category together? On the third hand (yes, computer scientists can have as many hands as it takes to describe the problem, as long as all hands, except for the first two, are virtual), connected products have unrelated star ratings. The last result is confusing, and you can leave the question open until you can afford to hire a subject-matter expert.

Because of the poor node assortativity by category, we expect a weird mixture of categories within any structural element—for example, within modularity-defined communities. Let's partition the network into communities and see how they are connected and named.

products.py
```
part = community.best_partition(gccs)
print("Modularity: {}".format(community.modularity(part, gccs)))
```

```
Modularity: 0.8241527716500038
```

The following statements create a list of lists of nodes in each community by collecting the nodes with the same partition ID. Function itertools.groupby() demands that the sequence is already sorted by the same key as would be used for grouping. In our case, the key is the partition ID, the second element of each tuple on the list returned by parts.items(), thus itemgetter(1). We will need the list later to auto-generate community labels.

products.py
```
groups = groupby(sorted(part.items(), key=itemgetter(1)), itemgetter(1))
community_labels = [list(map(itemgetter(0), group)) for _, group in groups]
subgraphs = [nx.subgraph(gccs, labels) for labels in community_labels]
```

We could use the previously constructed list of lists as a partition in nx.blockmodel(), but instead, we will utilize community.induced_graph(partition, graph), the blockmodeling tool from the community library:

products.py
```
induced = community.induced_graph(part, gccs)
induced.remove_edges_from(induced.selfloop_edges())
```

The induced graph usually has many self-loops because of copious connections between the nodes in the original network that belong to the same community. We remove the loops (on the highlighted line) to avoid clutter in the future network printout.

Name the Components

The new induced graph nicely reflects the macroscopic structure of the original product network. It has only eighteen nodes and twenty-nine edges. The nodes are nameless so far, and we need to give them names. Having no better source of labels than the product categories, we select the most popular category within each induced node as the node label. We need an auxiliary function to obtain the name of the dominant category in a community. The Sephora website reports category names as colon-separated hierarchical paths. To save space in the future printout, we keep only the last path component:

products.py

```
def top_cat_label(community_subgraph):
    items = [atts["category"] for _, atts
             in community_subgraph.nodes(data=True)]
    top_category = Counter(items).most_common(1)[0]
    top_label_path = top_category[0]
    return top_label_path.split(":")[-1]
```

Function `collections.Counter(sequence)` is an indispensable tool for counting occurrences of unique items in a `sequence`. It returns a dictionary-style `Counter` object with the method `Counter.most_common(n)` that reports a list of the `n` most popular items as `(label,count)` tuples (we only need the item label).

There may be several communities with the same dominant category in the network. If we blindly relabel them, their respective induced nodes will have the same label, and NetworkX will combine them into one node. To avoid unnecessary node merging, let's append the community ID to each label. The new labels look somewhat odd, but at least they are unique:

products.py

```
mapping = {comm_id: "{}/{}".format(top_cat_label(subgraph), comm_id)
           for comm_id, subgraph in enumerate(subgraphs)}
induced = nx.relabel_nodes(induced, mapping, copy=True)
```

At this point, our analysis is complete, but the data sponsor (the person or organization who ordered us the study) would rather see a nice picture than read a thousand barely decipherable labels. It's time to produce a picture. Function `graphviz_layout()` (*Harness Graphviz, on page 28*) attempts to find appropriate positions for the graph nodes, and `nx.draw_networkx()` draws the graph. The last function takes tons of parameters: you can customize edge and node colors, sizes, labels, and so on. You can save the resulting picture into a file, or display it on the screen, or both.

products.py

```
attrs = {"edge_color" : "gray", "font_size" : 12, "font_weight" : "bold",
         "node_size" : 700, "node_color" : "pink", "width" : 2,
         "font_family" : "Liberation Sans Narrow"}

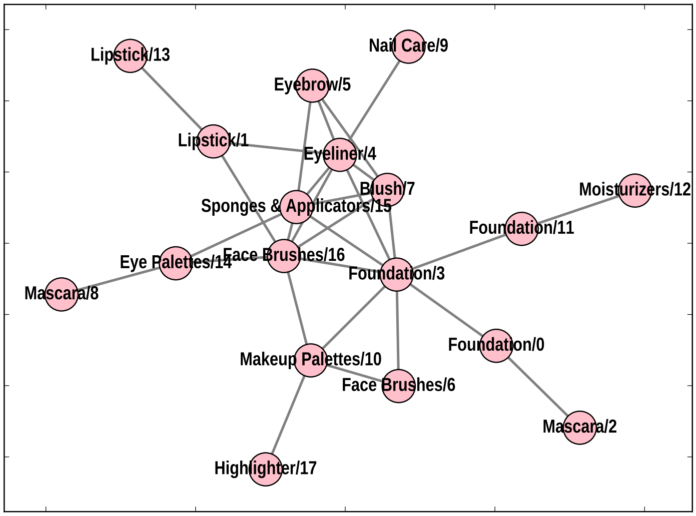
# Calculate best node positions
pos = graphviz_layout(induced)

# Draw the network
nx.draw_networkx(induced, pos, **dzcnapy.attrs)

# Adjust the extents
dzcnapy.set_extent(pos, plt)

# Save and show
dzcnapy.plot("ProductNetwork")
```

The output of the script is in the following figure. Since graphviz_layout() uses random numbers to calculate the best network layout, you will probably see a different picture when you execute the same code.



To understand the figure better, let's explore the degrees of the induced nodes with nx.degree(induced). The results are shown in the following table.

Node(s)	Degree
Foundation/14, Face Brushes/3	8
Makeup & Travel Cases/8	7
Blush/9	6
Eyeliner/13	5
Makeup Palettes/11	4
Eye Palettes/10, Eyebrow/16	3
Contour/17, Face Brushes/0, Foundation/15, Foundation/4	2
Moisturizers/7, Highlighter/5, Lipstick/1, Mascara/2, Mascara/6, Nail Care/12	1

At the top of the table, you can see the cosmetics essentials that are required for makeup but are not visible—namely, foundations and tools (brushes, cases, palettes). The most eye-catching tools are at the bottom of the table: mascaras, lipsticks, nail care tools, and highlighters. We can hypothesize that if a node is connected to (recommended to be "used-with") fewer neighbors, it is more specialized. The specialized nodes are at the periphery of the product network and depend on the more general nodes in the core.

Just like some other case studies presented in the book, the "products to projects" case is not limited only to the Sephora products. Given sufficient co-purchasing data, you can build a network of products, identify dense product communities, name them, and argue about possible reasons for their existence.

In the Next Part

The complex networks you have seen so far had a reasonably crisp structure. For any two nodes, you could say, with a fair degree of confidence, whether there was an edge incident to them or not. That is not how things work in real life.

In real life, there is almost always a degree of uncertainty involved in binary relationships. Alice, Bob, and Chuck may be friends, but Alice and Bob may be better friends than Alice and Chuck. A husky and a reindeer may be less likely to be bought together than a husky and a penguin. The uncertainty is a fact, and we need to know how to deal with it. In the next part, you will learn how to connect nodes in a fuzzy way based on potential similarity.

Part IV

Unleashing Similarity

Complex networks rarely have a crisp structure, whereby two nodes are unquestionably connected or not. The nodes are not always homogeneous, either. In this part, you will learn different ways of quantifying potential similarity between nodes and analyzing networks consisting of more than one class of nodes.

With these I have found nothing identical in any of the various books of Emblems which I have examined; indeed, I cannot say that I have met with anything similar.

➤ *Henry Green, English author*

CHAPTER 14

Similarity-Based Networks

This chapter uses Pandas, NumPy, SciPy.

Complex networks rarely have a crisp structure that shows whether two nodes are unquestionably connected. The nodes are not always homogeneous, either. However, sometimes two items are similar, and as a complex network analyst, you need a toolset for quantifying their similarity and converting similarities into network edges.

In this chapter, you will learn (or refresh your knowledge of) several similarity measures: Hamming distance, Manhattan distance, Pearson correlation, cosine distance, and generalized similarity. You will familiarize yourself with several real-world networks based on similarity.

Understand Similarity

Similarity-based networks emerge from the similarity of one or more attributes of objects represented by the network nodes. The type of objects and the number of attributes are limited only by the creativity of the network researchers. (This is not to say that your limitless imagination, rather than your experience, should guide your research.) The nodes may represent people (age, gender, language, skin color), products (price, color, shape, material), companies (industry, size, country, the form of ownership), and so on. It is your job to choose the "right" definition of similarity that at least does not contradict common sense.

Any quantitative measure of similarity has two aspects: what to measure and how to measure. In the case of similarity-based networks, the first aspect addresses the choice of significant nodes attributes, and the second aspect refers to transforming the attributes into distances.

Let's start with the first issue by looking at two very different similarity-based networks—an event network and a food network—and their node attributes (or the lack of them).

Creating New Attributes

When it comes to similarity-based networks, the most frustrating situation is when you want to use node attributes to calculate similarity, but the nodes have no attributes at all. The situation is not entirely desperate. Let's have a look at a dataset that furnishes no attributes, and build them from the ground up.

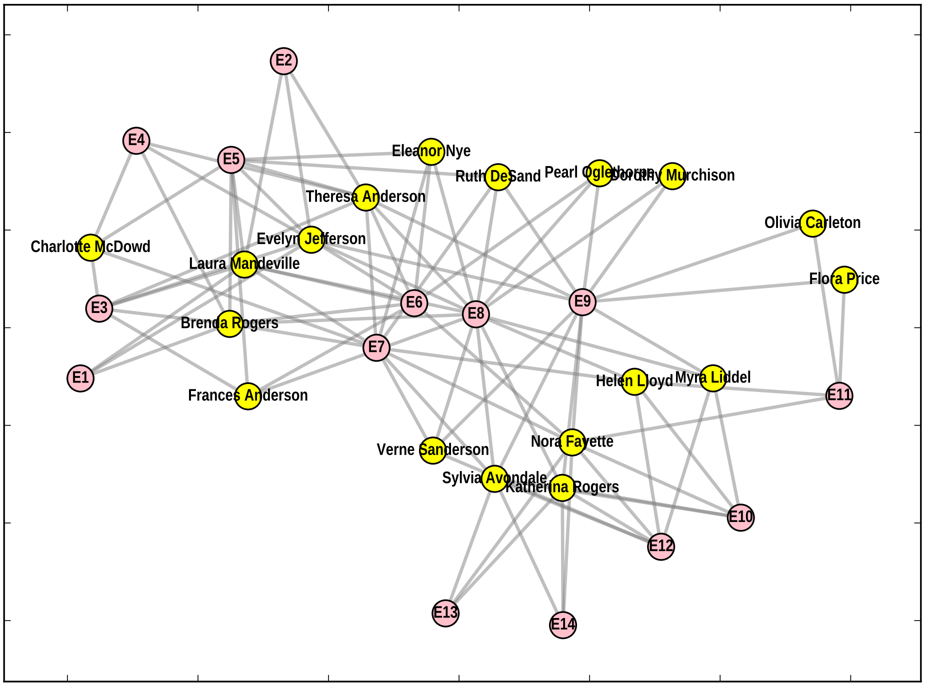
An event network is a similarity-based network where nodes represent formal or informal social events (book club meetings, political rallies, academic conferences, rock concerts, and the like) and edges depict their similarities. The nodes in an event network are usually not hard to identify, but the similarities may be subtle. Let's find out how to build an event network in five ways. (Incidentally, this chapter covers five types of similarity, dismisses one, and makes a promise to cover one more later.)

In the 1930s, five American ethnographers (Allison Davis and his colleagues) assembled a dataset of eighteen women in Natchez, Mississippi who attended fourteen social events over a nine-month period. Once published (*Deep South [DGG41]*), the dataset eventually became the foundation of a "canonical" "Southern Women" network. In fact, it is so respected in social network analysis that NetworkX has a special function `nx.davis_southern_women_graph()` for generating it. The figure on page 165 shows the network chart.

Suppose your goal is to transform the network of women and events into a network of events. Such transformations are called projections. You will learn more about projections in *Project Bipartite Networks,* on page 178; now, we will approach them informally.

Each event node in the original network is also a node in the event network. To calculate similarity, you must select relevant node attributes. Formally, the nodes have no attributes, aside from their arbitrarily assigned numerical labels. So, why not treat the identities of the women who attended an event as that event's attributes?

After you obtain the synthetic graph of the Southern women and the attended events `G1`, you should separate the nodes into the "women" and "events" subsets. In general, this operation may be hard, but in the Davis network, all event labels start with the capital letter *E*, followed by one or more decimal digits. If a node matches the regular expression *"E\d+"*, it



represents an event. All network neighbors of an event node (for example, *E1*) must be "women nodes," whose labels are accessible as the keys of the dictionary of neighbors. You can create a Pandas DataFrame that has only one column of ones, named after the event (*E1*) and indexed by the names of the attending women (in the case of *E1*: *Brenda Rogers*, *Laura Mandeville*, and *Evelyn Jefferson*).

southern_women.py

```
G1 = nx.davis_southern_women_graph()
attendees = [pd.DataFrame({event: 1}, index=list(women.keys()))
             for event, women in G1.edge.items() if re.match("E\d+", event)]
att_mtx = pd.concat(attendees, axis=1).fillna(0).astype(int)
print(att_mtx)
```

When you concatenate all fourteen DataFrames, Pandas performs the index alignment: each DataFrame is expanded as needed to place all data items with the same index in the same row. If the expansion results in gaps, the gaps are filled with a NaN—the NumPy designator for missing data. Replace the NaNs with zeros to complement the ones—and you've got a binary attribute set for each event node.

	E9	E7	E1	E2	E10	E5	E3	E12	E13	E11	E6	E14	E8	E4
Brenda Rogers	0	1	1	0	0	1	1	0	0	0	1	0	1	1
Charlotte Mc...	0	1	0	0	0	1	1	0	0	0	0	0	0	1
Dorothy Murc...	1	0	0	0	0	0	0	0	0	0	0	0	1	0
Eleanor Nye	0	1	0	0	0	1	0	0	0	0	1	0	1	0
Evelyn Jeffe...	1	0	1	1	0	1	1	0	0	0	1	0	1	1
Flora Price	1	0	0	0	0	0	0	0	0	1	0	0	0	0
Frances Ande...	0	0	0	0	0	1	1	0	0	0	1	0	1	0
Helen Lloyd	0	1	0	0	1	0	0	1	0	1	0	0	1	0
Katherina Ro...	1	0	0	0	1	0	0	1	1	0	0	1	1	0
Laura Mandev...	0	1	1	1	0	1	1	0	0	0	1	0	1	0
Myra Liddel	1	0	0	0	1	0	0	1	0	0	0	0	1	0
Nora Fayette	1	1	0	0	1	0	0	1	1	1	1	1	0	0
Olivia Carleton	1	0	0	0	0	0	0	0	0	1	0	0	0	0
Pearl Ogleth...	1	0	0	0	0	0	0	0	0	0	1	0	1	0
Ruth DeSand	1	1	0	0	0	1	0	0	0	0	0	0	1	0
Sylvia Avondale	1	1	0	0	1	0	0	1	1	0	0	1	1	0
Theresa Ande...	1	1	0	1	0	1	1	0	0	0	1	0	1	1
Verne Sanderson	1	1	0	0	0	0	0	1	0	0	0	0	1	0

Each column in the resulting matrix represents an event, each row accounts for a woman, and the number at an intersection is one of the eighteen event attributes—it indicates the presence (one) or absence (zero) of the woman at the event. You will learn in *Choose the Right Distance, on page 167*, how to connect the nodes to construct a similarity network.

Binarizing Existing Attributes

Dealing with node items that do not have attributes is tough. Items that do have attributes are much easier to handle. Just choose the attributes that are essential for your future network (you may want to use all available attributes or eliminate the insignificant features).

Do you remember the network of products in your pantry (*Explore Your Pantry, on page 120*)? You can use the data from the same USDA website to create a different kind of network.[1] For most food items, USDA provides the amounts of energy (in kcal), proteins, lipids, carbohydrates, fibers, and sugars (as well as minerals and vitamins) per serving. Each of the values is a potential continuous attribute of a future network node.

If the similarity measure you plan to use works only with binary attributes, you must first dichotomize (binarize) the attributes that don't fit. (In the case of the pantry project, all the attributes are in the wrong form.) Dichotomization can be accomplished either in plain Python or Pandas by comparing the value of an attribute with the mean or median value of the same attribute. Let's

1. ndb.nal.usda.gov/ndb/search

say the list protein contains the amounts of proteins in each food item. Then protein_bin is the dichotomized list:

```
import statistics
threshold = statistics.mean(protein)
# The choice of median ensures a balanced split!
# threshold = statistics.median(protein)
protein_bin = [p >= threshold for p in protein]
```

The Pandas solution involves converting the list to a Series unless the original data were in a Pandas format:

```
protein_ser = pd.Series(protein)
protein_bin = protein_ser >= protein_ser.mean()
# protein_bin = protein_ser >= protein_ser.median()
```

Whether you used original node attributes, dichotomized them, or inferred them from structural or other data, the next step in constructing a similarity-based network is to calculate distances between the nodes and convert them into weighted edges.

Choose the Right Distance

All similarity measures are numeric (usually on the scale from -1 to 1 or 0 to 1), so you must quantify any qualitative attributes before calculating similarities. Once quantified, the attributes can be thought of as coordinates of the object in a multidimensional coordinate space, where the number of dimensions equals the number of attributes. You can treat an object as a point in space, whose position is defined by the attributes. The similarity between two objects and the distance between the points representing the objects are complementary; the higher the distance, the smaller the similarity and vice versa.

Let's now have a look at some typical distance and similarity measures.

Hamming Distance

Let's suppose we want to build a network of objects that may have some binary features. A feature is either present or not, and if it is present, then the magnitude of the feature (if applicable) does not matter. The Hamming distance between two objects is the number of features that are present in one object, but not in the other, divided by the maximal number of features. The similarity, conversely, is the number of features jointly present or absent in both objects (again, divided by the maximal number of features). If two objects have an identical set of features, they are more similar than two objects whose features differ.

The Hamming distance definition can be extended to include categorical attributes. In this case, the distance between two objects is the number of attributes that are equal in both objects. The attributes do not even have to be quantitative.

Consider several vegetables with three attributes: shape, color, and starchiness. Some attributes are binary (for example, starchy) and some are qualitative (color).

Vegetable	Shape	Color	Starchy
Carrot	Conic	Orange	False
Corn	Conic	Yellowish	True
Potato	Round	Yellowish	True
Turnip	Round	Yellowish	False

Note that some attributes are binary and some are qualitative. Potatoes and turnips have two equal attributes (shape and color). The distance between them is 1/3, the similarity 2/3. Turnips and carrots are 1/3 similar, and potatoes and carrots are not similar at all (zero similarity), and so on. The following code fragment calculates Hamming similarity (complementary to the Hamming distance). The dataset for it has the same structure as G.node:

```
data = {
    "carrot" : {"shape": "conic", "color": "orange", "starchy": False},
    "corn" : {"shape": "conic", "color": "yellowish", "starchy": True},
    "potato" : {"shape": "round", "color": "yellowish", "starchy": True},
    "turnip" : {"shape": "round", "color": "yellowish", "starchy": False}
    }
# Collect attribute names for all vegetables
atts = set.union(*[set(x.keys()) for x in data.values()])

# Assume that each node has each attribute, but the values may differ
sim_2dlist = [[sum(data[v1][att] == data[v2][att] for att in atts)\
                                          / len(atts) for v1 in data]
              for v2 in data]
```

```
[[1.0, 0.3333333333333333, 0.0, 0.3333333333333333],
 [0.3333333333333333, 1.0, 0.6666666666666666, 0.3333333333333333],
 [0.0, 0.6666666666666666, 1.0, 0.6666666666666666],
 [0.3333333333333333, 0.3333333333333333, 0.6666666666666666, 1.0]]
```

The similarity between an item and itself is 1.

Python offers another way to calculate the Hamming distance: by calling the namesake function hamming() from the module scipy.spatial.distance. The benefit of using hamming() is the abstraction that the function creates: you or your

code reader do not need to understand exactly how the Hamming distance is calculated to benefit from it.

```
import scipy.spatial.distance as dist
sim_2dlist = [[1 - dist.hamming(list(data[v1].values()),
                                list(data[v2].values())) for v1 in data]
              for v2 in data]
```

Note that we subtract the distance from one to convert it to similarity. Incidentally, the module has another four dozen functions for similarity measurements, some of which you will see later.

The printout of sim_2dlist looks quite horrible and almost useless. Convert it to a NumPy array to add some order and enable vectorized operations:

```
sim_array = np.array(sim_2dlist)
```

```
❮ array([[ 1.        ,  0.33333333,  0.        ,  0.33333333],
         [ 0.33333333,  1.        ,  0.66666667,  0.33333333],
         [ 0.        ,  0.66666667,  1.        ,  0.66666667],
         [ 0.33333333,  0.33333333,  0.66666667,  1.        ]])
```

You still won't remember which column and row represent which vegetable, unless you convert the array to a Pandas DataFrame and supply human-readable labels:

```
sim_dataframe = pd.DataFrame(sim_array, columns=data, index=data)
```

```
❮           carrot      corn    potato    turnip
  carrot  1.000000  0.333333  0.000000  0.333333
  corn    0.333333  1.000000  0.666667  0.333333
  potato  0.000000  0.666667  1.000000  0.666667
  turnip  0.333333  0.333333  0.666667  1.000000
```

You can use this data to construct a weighted network and slice it if necessary, as explained in *Slice Weighted Networks*, on page 79.

The Hamming distance/similarity works best when future network nodes have many almost equally significant binary attributes whose presence/absence is roughly equally probable—such as the network of event nodes in *Creating New Attributes*, on page 164.

Manhattan Distance

If your objects have non-binary or non-categorical attributes, the Hamming distance is not applicable. The Manhattan distance is an extension of the Hamming distance for continuous attributes. It is defined mathematically as $d=|\Delta x_1|+|\Delta x_2|+\ldots+|\Delta x_N|$, where d is the distance between two nodes A and B, and $\Delta x_i=x_{Ai}-x_{Bi}$ is the difference between the values of their ith attribute.

In a human language, this means that the Manhattan distance between two objects is the sum of the differences of their attributes. For example, the distance between Pennsylvania station (x_{A1}=7th Avenue, x_{A2}=33rd Street) and the Metropolitan Opera (x_{B1}=Columbus Avenue, x_{B2}=64th Street) in Manhattan, New York, is |7-9|+|33-64|=2+31=33 blocks. (If you're not familiar with Manhattan, Columbus Avenue is another name for 9th Avenue. In either case, you also know now why the measure is called "Manhattan.")

To observe the Manhattan distance in action, let's have a look at the first five humans' heights and weights from the SOCR Data Dinov 020108 HeightsWeights dataset.[2]

```
hwdata = [[65.78, 112.99],
          [71.52, 136.49],
          [69.40, 153.03],
          [68.22, 142.34],
          [67.79, 144.30]]
```

SciPy provides function dist.cityblock(u,v) (because Manhattan is not the only city with blocks!) that takes two attribute vectors u and v and returns the Manhattan distance between them.

```
hw_array = np.array(hwdata)
five_ppl = np.array([[dist.cityblock(x, y) for x in hw_array]
                      for y in hw_array])
```

```
array([[  0.  ,  29.24,  43.66,  31.79,  33.32],
       [ 29.24,   0.  ,  18.66,   9.15,  11.54],
       [ 43.66,  18.66,   0.  ,  11.87,  10.34],
       [ 31.79,   9.15,  11.87,   0.  ,   2.39],
       [ 33.32,  11.54,  10.34,   2.39,   0.  ]])
```

You can define similarity as 1/five_ppl, max_distance-five_ppl, or in any other complementary or reciprocal way.

One major problem with the function dist.cityblock() is that it assumes that all attributes are comparable in range. This assumption does not hold in general, and it does not hold in the case of our five people in particular. Comparing weight to height is worse than comparing the proverbial apples to oranges! A workaround is to normalize each attribute by subtracting the smallest value and dividing by the range:

```
hw_range = hw_array.max(axis=0) - hw_array.min(axis=0)
hw_norm = (hw_array - hw_array.min(axis=0)) / hw_range
```

2. wiki.stat.ucla.edu/socr/index.php/SOCR_Data_Dinov_020108_HeightsWeights

```
array([[ 0.        ,  0.        ],
       [ 1.        ,  0.58691309],
       [ 0.63066202,  1.        ],
       [ 0.42508711,  0.73301698],
       [ 0.35017422,  0.78196803]])
```

```
five_ppl_norm = np.array([[dist.cityblock(x, y) for x in hw_norm]
                           for y in hw_norm])
```

```
array([[ 0.        ,  1.58691309,  1.63066202,  1.15810409,  1.13214225],
       [ 1.58691309,  0.        ,  0.78242489,  0.72101679,  0.84488073],
       [ 1.63066202,  0.78242489,  0.        ,  0.47255793,  0.49851977],
       [ 1.15810409,  0.72101679,  0.47255793,  0.        ,  0.12386394],
       [ 1.13214225,  0.84488073,  0.49851977,  0.12386394,  0.        ]])
```

In a normalized vector, the smallest attribute always has the value of 0 and the largest is always 1. The Manhattan distance between two normalized attribute vectors takes equal care of each attribute and is guaranteed to not be greater than 2×N. Treat the numbers in the array as edge weights—and you are one step away from a similarity network of persons based on their weight and height.

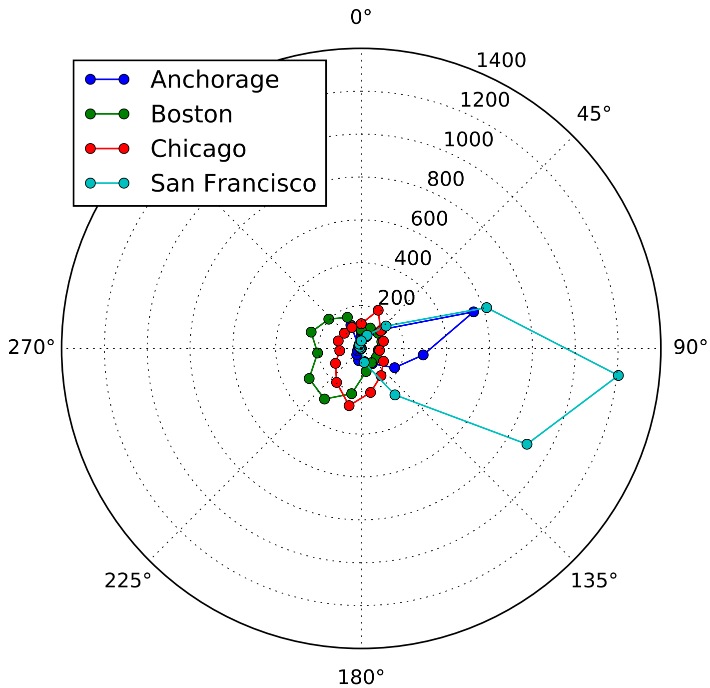
Euclidean Distance

In case you thought you did not know what the Euclidean distance was, here is a hint: it is the famous, though heavily tailored, Pythagoras' Trousers $d^2=\Delta x_1{}^2+\Delta x_2{}^2+\ldots+\Delta x_N{}^2$. The Euclidean distance is not particularly useful in CNA (except, perhaps, in geospacial networks) and is mentioned here simply because it is probably the most well-known distance measure.

Cosine Distance

The previous three distance/similarity measures treat nodes with N attributes as points in an N-dimensional space. Sometimes, it makes sense to treat attributes as directions—"lengthless" vectors. Consider a wind rose—a polar plot showing typical wind speed and direction distributions for a particular location. You can think of a wind rose as a set of sixteen floating point attributes representing the average wind speed in each direction with the 22.5° angular step (N, NNE, NE, ENE, E, and so on). The figure on page 172 shows the wind roses for four cornerstone American cities: Anchorage, Boston, Chicago, and San Francisco. The radius is the number of hours a year that the wind blows at speed from 7 to 12 mph *from* that direction.[3] So, who is the *real* Windy City?

3. www.meteoblue.com/en/weather/forecast/modelclimate/



In general, you can calculate the similarity between two geographical locations as some inverse distance between the attribute vectors. However, when it comes to winds, it may be more important to know where they blow from rather than how strong they are. In other words, the angle (angular distance) between the attribute vectors may be more useful than the linear distance.

The cosine distance is a measure of angular distance. It is defined as a complementary cosine of the angle between two attribute vectors x_A and x_B: $d=1-x_Ax_B/(\|x_A\| \times \|x_B\|)=1-\cos(x_A,x_B)$. The cosine similarity is the cosine itself. Naturally, it ranges from 1 (the angle is 0°, the vectors are parallel) through 0 (90°, the vectors are orthogonal and independent) to -1 (180°, the vectors are antiparallel).

The cosine distance emphasizes the similarity of shapes, spikes, and other patterns, rather than actual values. You can calculate it directly from the equation mentioned previously. (Remember that $\|x_A\|$ is the Euclidean length of x_A, and x_Ax_B is the scalar product of the two vectors.) Call the SciPy function cosine(u,v) to simplify your code and make it more abstract. The following code measures the cosine *similarity* (thus the 1-...) between the four cities, and it uses Pandas without further ado:

```
winds = {
    "Anchorage": (58,60,132,552,291,180,88,62,58,36,20,4,3,16,119,81),
    "Boston": (93,104,106,101,80,82,82,110,216,292,281,205,246,204,159,86),
    "Chicago": (115,195,122,109,86,120,157,210,273,196,139,101,113,106,
                107,115),
    "San Francisco": (35,67,156,616,1208,894,268,67,2,0,0,0,2,9,22,35)
    }
wind_cities_cosine = pd.DataFrame({y: [1 - dist.cosine(winds[x], winds[y])
                                       for x in winds] for y in winds},
                                  index=winds.keys())
```

```
               Anchorage    Boston   Chicago  San Francisco
Boston          0.408479  1.000000  0.884222       0.264567
Chicago         0.523189  0.884222  1.000000       0.381017
Anchorage       1.000000  0.408479  0.523189       0.791712
San Francisco   0.791712  0.264567  0.381017       1.000000
```

According to the printout, the two pairs of the most similar cities are Anchorage and San Francisco (on the West) and Boston and Chicago (on the East).

Pearson Correlation

One of the complications with the cosine similarity is that it is not invariant to shifts: it fails to detect small variations of attributes. Using the wind rose example, if you add another thousand hours a year to each direction in each considered city, the cosine similarities of each pair of cities will approach 1.0, making them all look the same. In other words, the cosine similarity formula overestimates similarity, which is not necessarily desirable.

Another angular similarity measure is the Pearson correlation, which some of you may know from statistics. It often goes by the name "correlation" without the reference to Karl Pearson. It is not affected by shifts.

The Pearson correlation is calculated using the same formula as for the cosine similarity, except that the attribute vectors are first translated by subtracting the mean m(x): $s=(x_A-m(x_A)) \times (x_B-m(x_B))/(\|x_A-m(x_A)\| \times \|x_B-m(x_B)\|)$.

SciPy provides the function stats.pearsonr(), which calculates both the correlation and its p-value as a tuple. If you're not sure what the p-value is, think of it as the measure of credibility of the reported correlation. If the p-value is less than 0.01, the correlation can be trusted. If the p-value is above 0.01, you should not take the correlation seriously even if it is high. The following code calculates the correlation-based similarities for the "wind cities."

```
wind_cities_pearson = pd.DataFrame({y: [stats.pearsonr(winds[x],
                                                       winds[y])[0]
                                        for x in winds] for y in winds},
                                   index=winds.keys())
```

```
                 Anchorage    Boston   Chicago  San Francisco
Boston           -0.482339  1.000000  0.234174      -0.524232
Chicago          -0.288015  0.234174  1.000000      -0.352705
Anchorage         1.000000 -0.482339 -0.288015       0.704106
San Francisco     0.704106 -0.524232 -0.352705       1.000000
```

An even more efficient solution is to convert the attribute dictionary `winds` into a Pandas `DataFrame` directly and then use the built-in method `.corr()`. The results are numerically the same, but the rows and columns are nicely sorted by the city names.

```
pd.DataFrame(winds).corr()
```

```
                 Anchorage    Boston   Chicago  San Francisco
Anchorage         1.000000 -0.482339 -0.288015       0.704106
Boston           -0.482339  1.000000  0.234174      -0.524232
Chicago          -0.288015  0.234174  1.000000      -0.352705
San Francisco     0.704106 -0.524232 -0.352705       1.000000
```

You'll see an efficient application of Pearson correlation similarity in Chapter 16, *Case Study: Building a Network of Trauma Types*, on page 185.

Generalized Similarity

The generalized similarity is a powerful recursive technique for measuring node similarities in bipartite networks. You need to learn more about bipartite networks to appreciate it. Let's postpone its introduction until *Compute Generalized Similarity*, on page 181.

In this chapter, you learned that if two items are not explicitly connected and don't happen to be at the same place at the same time, you can still connect them with a network edge if they are sufficiently similar. You learned about different types of similarity and how to calculate similarity based on node attributes. Very often, similarity-based networks are derived from bipartite networks—the networks that can be separated in two subnetworks in such a way that no two nodes in the same subnetwork are adjacent. You'll meet the bipartite networks in the next chapter.

Friendship is a single soul dwelling in two bodies.

➤ *Aristotle, Greek philosopher*

CHAPTER 15

Harnessing Bipartite Networks

Bipartite networks, also known as two-mode networks, are a mechanism for representing relationships between items that belong to two or more classes or parts (such as students and professors or airlines and airports). Many networks previously seen in this book are bipartite.

In this chapter, you will learn how to check if a network is bipartite, assign the nodes to the respective parts, and convert weighted or unweighted bipartite networks into weighted one-part networks (the latter operation is called "projection").

You Don't Have to Be Bipartite—Even If You Can

Being bipartite is both a topological property of a network and the way the part attributes are assigned to the nodes. Some networks —such as a ring with an odd number of nodes—cannot possibly be treated as bipartite. No matter how you label the nodes, there will always be two adjacent nodes that belong to the same part. Conversely, a ring with an even number of nodes can be bipartite —but only if odd and even nodes belong to different parts.

It may be hard to believe, but you already built your first bipartite network—it was the network of food items and nutrients in *Draw Your First Network with Paper and Pencil,* on page 6. The network has two namesake classes of nodes: food items (beef, spinach, and so on) and nutrients (vitamin C, magnesium, and so on). Each food item node is adjacent only to nutrient nodes, and each nutrient node is adjacent only to food item nodes. An edge always connects nodes that belong to different parts.

Work with Bipartite Networks Directly

Many networks you met in the book are bipartite. Before we look into the NetworkX functions and tools for the bipartite networks, let's first revisit them.

Examples of Bipartite Networks

The following table contains the list of the some bipartite networks mentioned in this book so far.

Name	Location	Part 1	Part 2
Introductory toy network	on page 6, on page 21	Food items	Nutrients
"Panama papers"	on page 101	Entities and intermediaries	Officers
Products in your pantry	on page 120, on page 166	Food items	Ingredients
LiveJournal	on page 141	Users	Interest terms
Southern women	on page 164	Women	Events

Except for the "Panama papers," each network has nodes of two types. (You can conditionally put the "Panama" entities and intermediaries in one part, but the resulting network is still not strictly bipartite. It is tripartite, as explained in the following sidebar.)

How About More Parts?

You have seen unipartite networks where any node can be connected to any node. You have seen bipartite networks where nodes link only to nodes from the other part. The concept of network parts can be readily extended to the case of k-partite networks with `k` parts, as long as two adjacent nodes do not belong to the same part. An example of a tripartite network is a network of LiveJournal users (part 1), communities (part 2), and interest terms (part 3). A user belongs to a community (a 1↔2 type edge); a user is interested in a term (a 1↔3 type edge); and a community declares a term as an interest (a 2↔3 type edge).

You can build bipartite networks naturally because quite often real-world datasets already include nodes of more than one type. However, they are not easy to analyze and interpret. For starters, even such a basic measure as node degree may be of questionable use in a bipartite network. Indeed, does a company node with one hundred adjacent employee nodes have the same degree as an employee node with one hundred adjacent company nodes? In the former case, we are talking about a typical medium-size business. In the

latter case, the employee is probably a freelancer trying to make their one hundred clients happy. Different types of centralities, paths, cores, cliques, modularity-based communities, and other measures and structural elements of bipartite networks may be similarly incomparable and even meaningless.

Basic Bipartite Functions

To check whether network G is bipartite, call the predicate function nx.is_bipartite(G). If the function returns False, skip the rest of the chapter. Wait, don't. We will use the pickled network of foods and nutrients nutrients.pickle (created in *Read a Network from a CSV File, on page 21*), which is known to be bipartite.[1]

bipartite.py
```
from networkx.algorithms import bipartite
N = pickle.load(open("nutrients.pickle", "rb"))
print(bipartite.is_bipartite(N))
```

```
❮ True
```

Note most of the bipartite functions come from the module nx.algorithm.bipartite, which you must correctly import.

Function bipartite.sets() splits the nodes of a bipartite network into two parts (and returns two node sets). The function does not look at the node attributes. The separation it performs is based purely on the network topology. It is your responsibility to recognize the meaning of each part. For example, you can check which set contains the vitamin *C*. The same set must contain all other nutrients.

bipartite.py
```
bip1, bip2 = bipartite.sets(N)
print("C" in bip1, "C" in bip2)
```

```
❮ False True
```

You can use the following code fragment to initialize the two sets without second-guessing which part contains which nodes:

bipartite.py
```
foods, nutrients = (bip2, bip1) if "C" in bip1 else (bip1, bip2)
print(foods, nutrients)
```

```
❮ {'Spinach', 'Beans', 'Poultry', 'Veg Oils', 'Green Leafy Vegs', 'Cheese',
  'Asparagus', 'Potatoes', 'Fatty Fish', 'Carrots', 'Beef', 'Liver',
  'Seeds', 'Mushrooms', 'Eggs', 'Broccoli', 'Wheat', 'Whole Grains',
  'Pumpkins', 'Tomatoes', 'Kidneys', 'Legumes', 'Yogurt', 'Milk', 'Nuts',
  'Shellfish'} {'Thiamin', 'Folates', 'B6', 'E', 'Mn', 'Se', 'B12', 'D',
  'A', 'Riboflavin', 'C', 'Zn', 'Cu', 'Niacin', 'Ca'}
```

1. pragprog.com/titles/dzcnapy/source_code

It is possible to analyze two-mode networks directly (*Networks: An Introduction [New10]*, *Exploratory Social Network Analysis with Pajek (Structural Analysis in the Social Sciences) [NMB11]*, *Analyzing Social Networks [BEJ13]*), and networkx.algorithms.bipartite provides a variety of functions for such analysis. The functions are mostly direct counterparts of the unipartite namesake brethren: bipartite.density(), bipartite.degrees() (cf. nx.degree()), bipartite.clustering(), bipartite.closeness_centrality(), bipartite.degree_centrality(), bipartite.betweenness_centrality(), and some generator functions, including bipartite.random_graph(). It is also customary to convert bipartite networks into unipartite networks by projecting on one of the constituent parts.

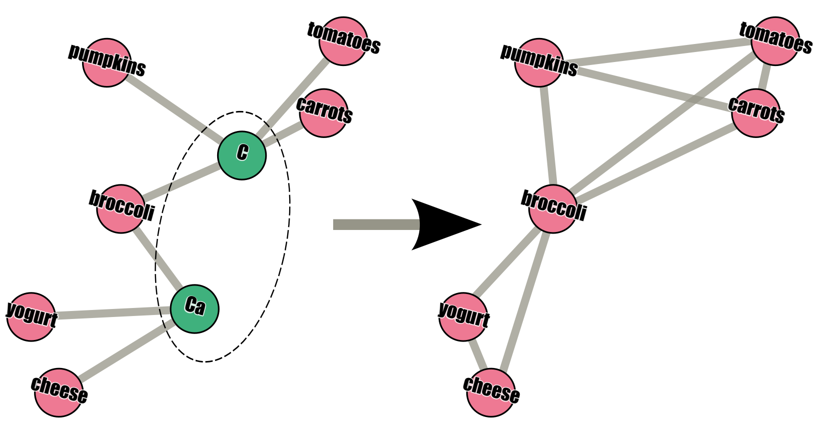
Project Bipartite Networks

This section uses SciPy, Pandas.

You can project a bipartite network two ways: by keeping the nodes of part 1 and removing the nodes of part 2, and the other way around. The nodes that survive the projection are called the "bottom" nodes; the nodes that are removed are known as the "top" nodes.

The projection operation transforms the original bipartite graph G into an induced graph F by projecting G onto the bottom nodes. Graph F contains only the bottom nodes, and two bottom nodes in F are adjacent to each other if and only if they are adjacent to the same top node in G.

The following figure shows a fragment of the bipartite network of foods and nutrients before (left) and after the projection. The nodes *C* and *Ca* represent nutrients; they are the top nodes. All other nodes represent foods; they are the bottom nodes. Note that all food nodes connected to the same nutrient node in the original network form a clique in the induced network—the clique of foods providing that nutrient.



Function bipartite.projected_graph((G,nodeset)) projects a bipartite graph G onto the nodes nodeset (the nodes must exist in G and belong to the same part).

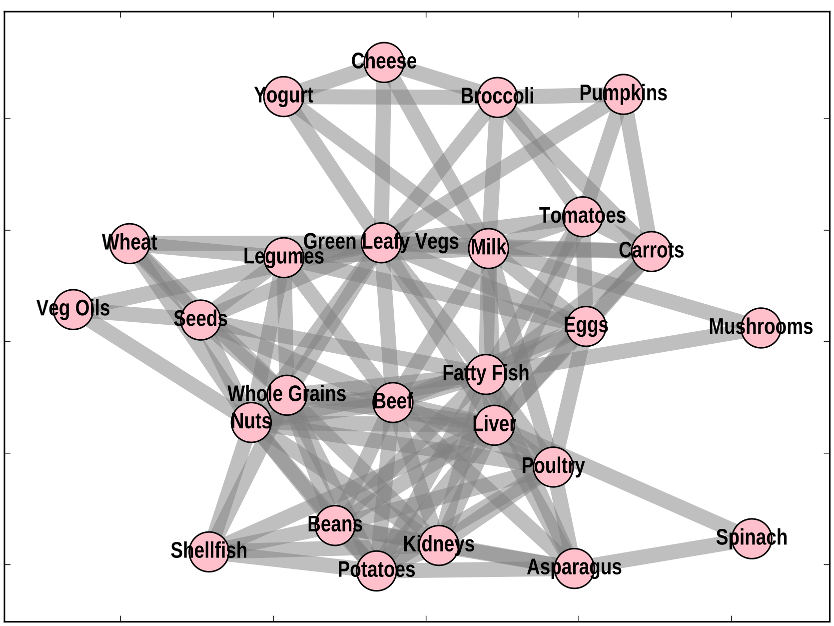
bipartite.py
```
n_graph = bipartite.projected_graph(N, nutrients)
f_graph = bipartite.projected_graph(N, foods)
```

The resulting network is undirected, unweighted, and unipartite. (And, by the way, it is a product network.) You can calculate degrees, centralities, and path lengths; extract cliques, cores, and communities; and perform any other complex network analysis of it. The network still contains some knowledge of the connecting nutrients, but the knowledge is implicit. Just like a geometric projection, a network projection is lossy and irreversible (one cannot reconstruct the original bipartite network from one of its projections).

There may be more than one top node connecting a pair of bottom nodes in the same network. You can assign weights to the induced edges to reflect the connection strength by calling the function bipartite.weighted_projected_graph(G,nodeset,ratio). (The last parameter controls whether the weights are absolute or relative.)

bipartite.py
```
fw_graph = bipartite.weighted_projected_graph(N, foods, True)
```

The following figure shows the weighted induced network of food items connected by the similarity in terms of provided nutrients. It is still undirected and unipartite.



Edge width in the figure represents the weights, and the weights represent the similarity between the nodes. (The more shared nutrients they had in the original network, the higher the similarity.) The induced network is a similarity-based network, and everything you read about such networks in Chapter 14, *Similarity-Based Networks*, on page 163 applies to it, too. Moreover, the network is based on *Hamming Distance*! You may wonder if you can use other distance definitions. Yes, you can—but on your own, without much help from NetworkX.

As an exercise, let's build a network based on the *Pearson Correlation*. Start by calculating the bi-adjacency matrix. A bi-adjacency matrix is like an *Incidence Matrix*, except that the matrix rows and columns represent the top and bottom nodes, respectively. (You have to tell NetworkX which nodes are top and which are bottom by passing the list of bottom nodes as the second parameter.) For each pair of rows, compute the Pearson correlation and arrange the results into a square Pandas DataFrame food.

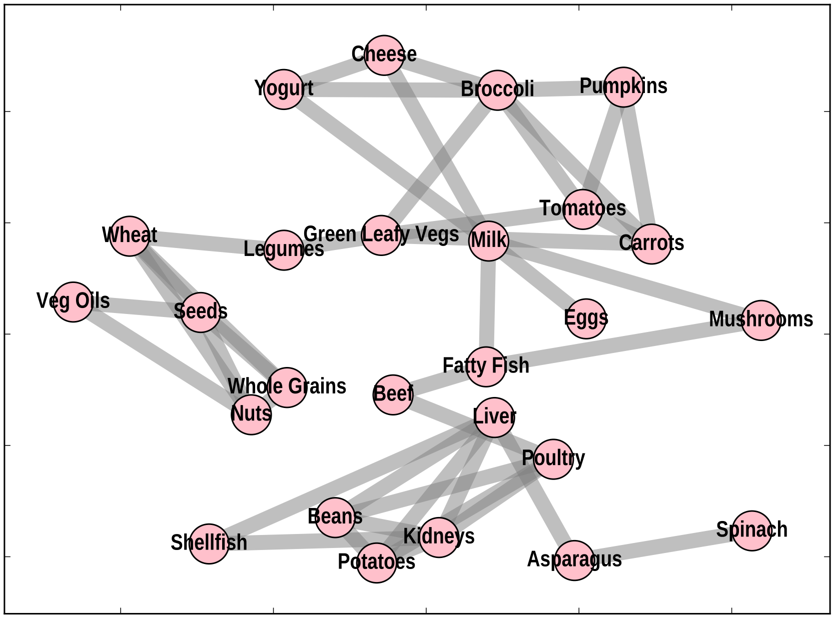
bipartite.py
```
adj = bipartite.biadjacency_matrix(N, f_graph).toarray()
foods = pd.DataFrame([[stats.pearsonr(x, y)[0] for x in adj]
                      for y in adj], columns=f_graph, index=f_graph)

SLICING_THRESHOLD = 0.375
stacked = foods.stack()
edges = stacked[stacked >= SLICING_THRESHOLD].index.tolist()
f_pearson = nx.Graph(edges)
```

The matrix contains the similarities between the bottom nodes with respect to the connectivity to the top nodes. Some similarities are negative; at the very least, you must not convert them into edges. In fact, let's discard as many potential edges as possible, as long as the network remains connected. In this example, the slicing threshold of 0.375 was chosen by trial and error (see details on page 80). Note that this value is still statistically low: one would hardly consider 0.375 a significant correlation!

After slicing, arrange the surviving edges into a network and plot it. (Call function nx.from_pandas_dataframe(df,source,target) from *Adjacency Matrix, the Pandas Way*, on page 73 if you want to create a weighted network.) The figure on page 181 shows the correlation-based network of foods. Compared to the previous figure, the network has the same nodes (and in the same locations), but fewer edges.

You can extend the proposed algorithm to project bipartite networks using Euclidean, cosine, and any other reasonably defined distance measure. All of them have a subtle problem: they assume that all the top nodes are independent, and adjacency to each of them is equally important. Sometimes this assumption is correct; sometimes it is not. Consider the nutrients from our



dataset. It includes vitamins B6 and B12, niacin (also a vitamin B), and riboflavin (yet another kind of vitamin B). All projection algorithms considered so far treat these four nutrient nodes separately. If a food item provides B6 but not B12 and another item provides B12 but not B6, they are not considered similar. But they would be—if you merged the four specific vitamin B nodes into one umbrella node.

If you have a strong reason to believe that some top nodes are more similar to each other than the others, you may want to compute the so-called generalized similarity.

Compute Generalized Similarity

This section uses generalized.

Traditionally, two bottom nodes are considered similar if they are adjacent to the *same* top node or to a set of *same* top nodes, even though the *sameness* may be too strict a requirement. *Kovacs [Kov10]* proposed to weaken the definition of similarity. The new measure, dubbed "generalized similarity," treats two bottom nodes as similar if they are adjacent to *similar* top nodes. But who decides whether two top nodes are similar? It is the reflexive definition of generalized similarity itself: two top nodes are similar if they are adjacent to *similar* bottom nodes. In fact, the algorithm for calculating the generalized

similarity does not care whether a node is top or bottom. It splits a bipartite network into two parts and reports the similarities for each node pair in each class with respect to the nodes in the other class.

The generalized similarity is computed iteratively. The algorithm repeatedly calculates the pairwise Pearson correlations of the nodes in each part of the network, gradually transforming the original Euclidean coordinate system into an affine coordinate system. (The angles between the affine coordinate axes, in general, are not right.) The angles between the axes that represent similar items become more acute; the angles between dissimilar items become more obtuse. (Remember that originally all items are considered independent, that's why all angles were right.) The Pearson correlation calculated in the new deformed coordinate system better reflects the similarities of the nodes in each network part.

The process is repeated until the affine coordinate system stabilizes and stops morphing. The iterations may take considerable time. You can put a cap either on the maximal number of iterations or the minimal deformation magnitude at each iteration. A perfect solution for a large network (1,000 or more nodes) is usually infeasible, anyway.

Module `generalized` implements the Kovacs algorithm. You can download the module from GitHub[2] or the book's website[3] as `generalized.py`.

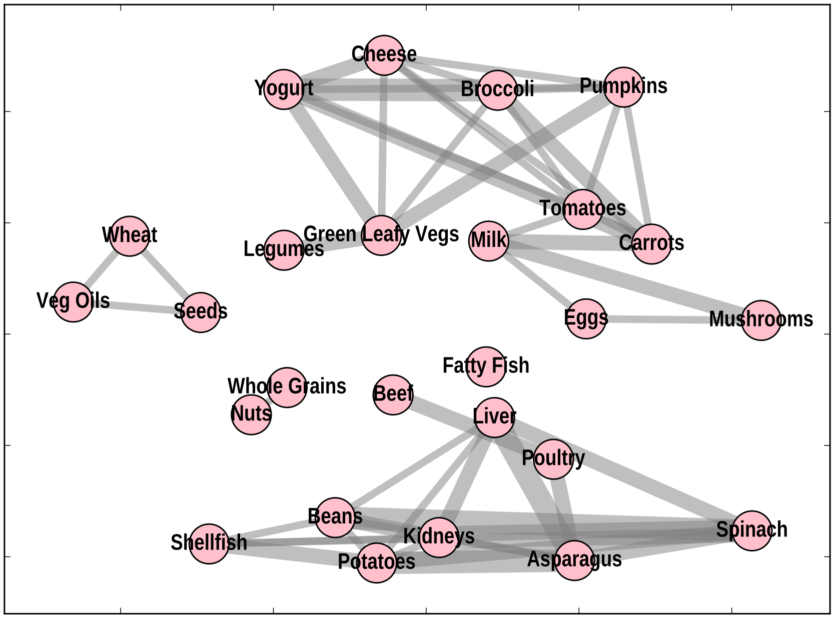
Module `generalized` provides only one function `generalized_similarity(G, min_eps=0.01, max_iter=50)`. The function takes a bipartite network and up to two loop termination hints and returns a tuple of four values: two unipartite, undirected, weighted similarity networks; the attained precision; and the number of completed iterations. Start the analysis by calling the function:

bipartite.py

```
from generalized import generalized_similarity
bip1, bip2, eps, n_iter = generalized_similarity(N, min_eps=0.001,
                                                 max_iter=100)
foods, nutrients = (bip1, bip2) if "C" in bip2 else (bip2, bip1)
SLICING_THRESHOLD = 0.9
foods.remove_edges_from((n1, n2) for n1, n2, d in foods.edges(data=True)
                        if d['weight'] < SLICING_THRESHOLD)
```

The rest of the script identifies and slices the network of interest, then truncates the "weak" edges. The figure on page 183 shows the network of food items based on the generalized similarities.

2. github.com/dzinoviev/generalizedsimilarity
3. pragprog.com/titles/dzcnapy/source_code



The most prominent difference between the latter network and the previous two attempts is the redistribution of graph density from the "center" (OK, we know that networks do not have centers!) to the "periphery"—to the extent that *Fatty Fish* became an isolate. You can merge it back into the giant component by playing with the `SLICING_THRESHOLD` at the expense of having less structure in the other parts of the network if you want.

As a free byproduct of the projection, you got a network of the nutrients, `nutrients`. The twin networks in the generalized similarity analysis problems may have considerable size and waste the precious memory of your computer. If you don't plan to use them, tell Python: `del nutrients`.

Bipartite network and networks that consist of more than two parts are much more common in life than one may be inclined to think. Treating a network as bipartite gives you additional CNA tools (various types of projections), adds another dimension to your projects, and empowers you to discover unexpected dependencies between the nodes.

In the next chapter, you will see how to build a network of something seemingly totally unrelated to networks—psychological trauma types. Not only is it cool, but it also helps to diagnose psychiatric disorders!

Sometimes, as she sat alone in the arm-chair in her room, she would begin laughing and crying at the same time, with a sort of tearless grief, or else relapse into convulsions, and scream out dreadful, incoherent words in a horrible voice. It was the first dire sorrow which she had known in her life, and it reduced her almost to distraction.

➤ *Leo Tolstoy, Russian writer*

CHAPTER 16

Case Study: Building a Network of Trauma Types

This chapter uses Pandas, NumPy, community, generalized.

A typical dataset for bipartite network construction consists of objects and their properties, such that each object has several properties and each property is found in several objects. In this case study, objects are subjects of a mental trauma study (suitably anonymized), and properties are their trauma types. You will learn how to derive a network of trauma type (or other properties) based on the subjects' experience.

Embark on Psychological Trauma

Exposure to traumatic events is quite common among children and adolescents. One notable challenge facing trauma researchers is understanding the nature and importance of the co-occurrence of exposure to different types of psychological trauma. You may wonder if complex network analysis is the right (or a right) tool for this task.

CNA has been indeed widely used in medical and public health research in the last decade (see, for example, work by *Nicholas Christakis [CF09]* and *A.-L. Barabási [BGL11]*, and the chapters on network epidemiology in *Network Science. Theory and Applications [Lew09]* and *Networks, Crowds, and Markets. Reasoning about a Highly Connected World [EK10]*). However, the focus of those studies was mainly on social networks or gene networks. We can go the extra mile and look at the network of diagnoses—the trauma types—defined by their similarity with respect to exposure. Such a network, if constructed, could be used to classify trauma types, which would hopefully improve the quality of trauma diagnostics and treatment. In fact, the network

has been constructed, and it indeed substantially improved the quality of diagnostics (*Network Analysis of Exposure to Trauma and Adverse Events in a Clinical Sample of Children and Adolescents [HSZL17]*). You have all the necessary skills not only to reproduce the process but also to look at several possible trauma networks.

The goal of this case study is not just to show you how to read CSV files or construct similarity-based networks. After all, you have been reading about that stuff for almost two hundred pages. You will see that, given the same data, you can transform it into different networks and perhaps even come to different conclusions.

Read the Data, Build a Bipartite Network

As always, the script starts with a fat chunk of `import` statements.

jri_code.py
```
import pandas as pd
import numpy as np
import networkx as nx
from networkx.algorithms.bipartite import sets, weighted_projected_graph
from networkx.drawing.nx_agraph import graphviz_layout
import scipy.spatial.distance as dist
from scipy.stats import pearsonr
import community
import generalized
import dzcnapy_plotlib as dzcnapy
import matplotlib.pyplot as plt
```

Boston's Justice Resource Institute generously provided the dataset for this project.[1] You can find it in the file `jri_data.csv`. The file is a correctly formatted CSV table with standard delimiters and a header row at the top. Each of the nineteen columns represents a trauma type.

jri_code.py
```
matrix = pd.read_csv("jri_data.csv")
print(matrix.columns, matrix.shape)
```

```
❮ Index(['SEXUAL_ABUSE', 'SEXUAL_ASSAULT', 'PHYSICAL_ABUSE', 'PHYSICAL_ASSAULT',
       'PSYC_MALTX', 'NEGLECT', 'DOMESTIC_VIOLENCE', 'WAR', 'WAR_NOT_US',
       'MEDICAL_TRAUMA', 'INJURY_ACCIDENT', 'NATURAL_DISASTER', 'KIDNAP',
       'TRAUMATIC_LOSS', 'FORCED_DISPLACEMENT', 'IMPAIRED_CAREGIVER',
       'EXT_INTERPER_VIOLENCE', 'COMMUNITY_VIOLENCE', 'SCHOOL_VIOLENCE'],
      dtype='object') (618, 19)
```

1. jri.org

The trauma names are reasonably self-explanatory, except for PSYC_MALTX ("physical maltreatment"), WAR_NOT_US ("war outside the USA"), and EXT_INTERPER_VIOLENCE ("extended interpersonal violence").

Each row represents one patient (or subject, as they used to refer to participants in social and behavior studies not so long ago). The original dataset has been already anonymized to preserve the patients' privacy. The JRI staff replaced each patient's name with a unique integer number. Since in this study we do not care about the patients' identity at all, the JRI identifiers have been removed altogether. All we know is that a patient of an unknown age and gender has been exposed to a set of psychological traumas at an unknown time. Respectively, the values of the DataFrame matrix are zeros and ones (in the floating-point format), depending on whether the patient in a row was diagnosed with the trauma in a column or not.

Let's build the network of the trauma types four different ways: from Hamming similarity, cosine similarity, Pearson correlation, and generalized similarity. At the moment, you know that each method evaluates similarity in its way and none of the four methods seems to have a clear advantage over the other three. If you randomly commit yourself to one of the methods, you may end up with an inaccurate, distorted, or even incorrect network. You will be able to select the most efficient analysis tool by the end of this chapter.

The first and last networks are induced, so we need a bipartite network patients_traumas of patients and trauma types first. We will construct the other two networks directly from the matrix. The next code fragment prepares the bipartite network and double checks if it is indeed bipartite. Note that the matrix is the bi-adjacency matrix of the network of interest (explained on page 180).

jri_code.py

```
# Make a multi-index of patients+traumas
stacked = matrix.stack()
# Select the patients who _have_ traumas
edges = stacked[stacked > 0].index.tolist()
patients_traumas = nx.Graph(edges)
print(nx.is_bipartite(patients_traumas))
```

```
❮ True
```

The bipartite network is just an intermediate milestone for this project. There is no point in visualizing or analyzing it.

Build Four Weighted Networks

Now that you have all the preprocessed data (DataFrame matrix and network patients_traumas), you can transform it into four weighted networks.

Each column in the matrix is a 618-dimensional vector of binary properties of the future trauma node: the property of being diagnosed in patient 0; the property of being diagnosed in patient 1; and so on. Surely, two trauma types are similar if the vectors are similar in some sense. Once the similarities of each pair of vectors are known, the process of network construction is straightforward and can be implemented as a set of functions—at least for the cosine and Pearson distances.

jri_code.py

```
def similarity_mtx(biadj_mtx, similarity_f):
    """
    Convert a bi-adjacency matrix to a similarity matrix,
    based on the distance measure
    """
    similarity = [[similarity_f(biadj_mtx[x], biadj_mtx[y])
                   for x in biadj_mtx] for y in biadj_mtx]
    # Discard the main diagonal of ones
    similarity_nodiag = similarity * (1 - np.eye(biadj_mtx.shape[1]))
    similarity_df = pd.DataFrame(similarity_nodiag,
                                 index=biadj_mtx.columns,
                                 columns=biadj_mtx.columns)
    return similarity_df
```

The function similarity_mtx(biadj_mtx, similarity_f) takes the bi-adjacency matrix and a similarity measure (a two-argument function that returns the similarity of its parameters) and returns the similarity matrix. The matrix always has ones on the main diagonal because each node is similar to itself. The function removes the main diagonal, which otherwise would result in a bunch of self-loop edges.

jri_code.py

```
def similarity_net(sim_mtx, threshold=None, density=None):
    """
    Convert a similarity to a sliced similarity network
    """
    stacked = sim_mtx.stack()
    if threshold is not None:
        stacked = stacked[stacked >= threshold]
    else:
        count = int(sim_mtx.shape[0] * (sim_mtx.shape[0] - 1) * density)
        stacked = stacked.sort_values(ascending=False)[:count]
    edges = stacked.reset_index()
    edges.columns = "source", "target", "weight"
```

```
    network = nx.from_pandas_dataframe(edges, "source", "target",
                                       edge_attr=["weight"])
    # Some nodes may be isolated; they have no incident edges
    network.add_nodes_from(sim_mtx.columns)
    return network

DENSITY = 0.35
```

The function similarity_net(sim_mtx, threshold=None, density=None) slices the similarity matrix and converts the surviving entries into the edges of the induced network. Depending on the chosen similarity measure, the interpretation of the edge weight differs. It is not fair to use the same slicing threshold for two similarity matrices computed with different distances and expect them to be comparable. That's why the function performs slicing based either on the slicing threshold (for the networks that are based on the same distance) or desired network density. Considering that your four networks emerge from four different distance measures, you must use the density-based mechanism. The density of 0.35 seems to produce a nice collection of networks, but you are encouraged to experiment with it.

Two trauma nodes are cosine similar if the angle between their vectors (*Cosine Distance*, on page 171) is small and the cosine of the angle is large.

jri_code.py
```
def cosine_sim(x, y):
    return 1 - dist.cosine(x, y)

cosine_mtx = similarity_mtx(matrix, cosine_sim)
cosine_network = similarity_net(cosine_mtx, density=DENSITY)
```

Two trauma nodes are Pearson similar if their vectors are positively correlated (*Pearson Correlation*, on page 173). An added benefit of Pearson correlation is that it comes with a p-value. If you want to consider only statistically significant correlations (say, with the p-value<0.01), you can modify the similarity function appropriately.

jri_code.py
```
def pearson_sim(x, y):
    return pearsonr(x, y)[0]

pearson_mtx = similarity_mtx(matrix, pearson_sim)
pearson_network = similarity_net(pearson_mtx, density=DENSITY)

# Shall we discard the statistically insignificant ties?
def pearson_sim_sign(x, y):
    r, pvalue = pearsonr(x, y)
    return r if pvalue < 0.01 else 0

pearson_mtx_sign = similarity_mtx(matrix, pearson_sim_sign)
pearson_network_sign = similarity_net(pearson_mtx_sign, density=DENSITY)
```

The Hamming and generalized similarity networks are weighted and complete projections of the patients_traumas bipartite network. They already contain all edges, and your job is to remove the "weak" edges. The function slice_projected(net, threshold=None, density=None) is an equivalent of similarity_net(sim_mtx, threshold=None, density=None) and removes the edges that have small weight or make the network too dense.

jri_code.py
```
def slice_projected(net, threshold=None, density=None):
    """
    Slice a projected similarity network by threshold or density
    """
    if threshold is not None:
        weak_edges = [(n1, n2) for n1, n2, w in net.edges(data=True)
                      if w["weight"] < threshold]
    else:
        count = int(len(net) * (len(net) - 1) / 2 * density)
        weak_edges = [(n1, n2) for n1, n2, w in
                      sorted(net.edges(data=True),
                             key=lambda x: x[2]["weight"],
                             reverse=True)[count:]]
    net.remove_edges_from(weak_edges)
```

Two trauma nodes are Hamming similar if the trauma types have been frequently observed together in the same patients (*Hamming Distance*, on page 167).

jri_code.py
```
net1, net2 = sets(patients_traumas)
_, traumas = (net1, net2) if "WAR" in net2 else (net2, net1)
hamming_network = weighted_projected_graph(patients_traumas,
                                           traumas, ratio=True)
slice_projected(hamming_network, density=DENSITY)
```

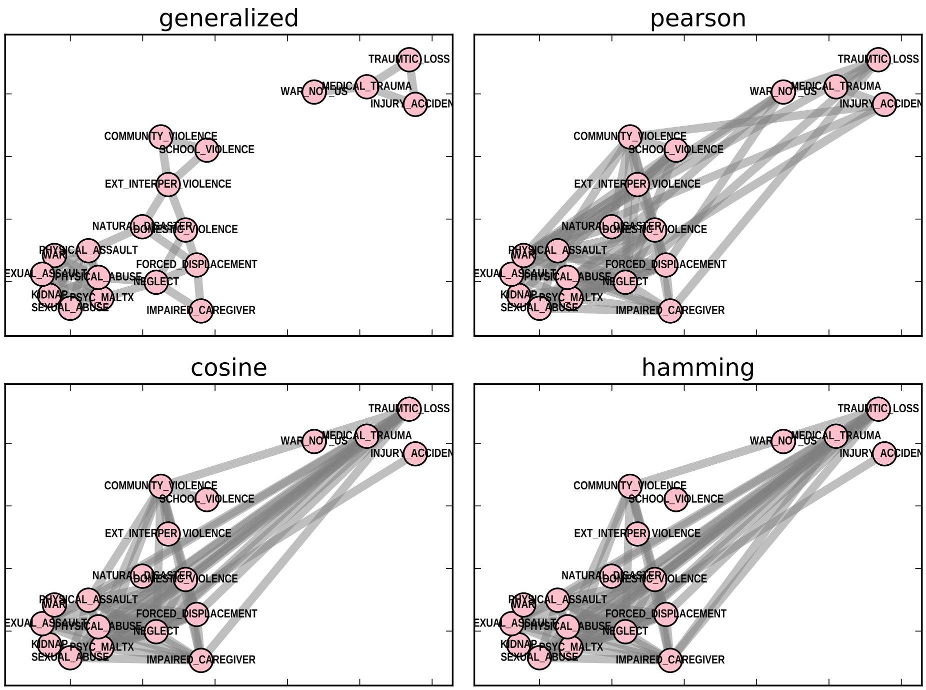
Two trauma nodes are generally similar if the trauma types have frequently been observed in similar patients (*Generalized Similarity*, on page 174). This piece of code incidentally also generates a similarity network of the patients, which you do not need for this project.

jri_code.py
```
net1, net2, eps, n = generalized.generalized_similarity(patients_traumas)
_, generalized_network = (net1, net2) if "WAR" in net2 else (net2, net1)
slice_projected(generalized_network, density=DENSITY)
generalized_network.remove_edges_from(generalized_network.selfloop_edges())
```

Congratulations! You aspired to compute a network of trauma types. Instead, you got four (or five, if you count two versions of the Pearson network as two networks). But which one is the best and which are on the chopping block?

Plot and Compare the Networks

All four networks have the same number of nodes and similar density (and a similar number of edges), which makes them very easy to compare. The following picture shows the charts of all four networks.



The difference between the tidy generalized similarity network cleanly separated into two components, and its brethren, is striking. You can still see some structure in the Pearson network, but the other two graphs are nearly random.

The numerical experiment with community structure extraction (see *Outline Modularity-Based Communities, on page 136*) confirms: the first network has an acceptable modularity of 0.47 and three network communities. The other networks are not very modular.

jri_code.py

```
networks = {
    "generalized" : generalized_network,
    "pearson" : pearson_network_sign,
    "cosine" : cosine_network,
    "hamming" : hamming_network,
    }
```

```
partitions = [community.best_partition(x) for x in networks.values()]
statistics = sorted([
        (name,
         community.modularity(best_part, netw),
         len(set(best_part.values())),
         len(nx.isolates(netw))
         ) for (name, netw), best_part in zip(networks.items(), partitions)],
                    key=lambda x: x[1], reverse=True)
```

The following table shows the modularity-related statistics of the four networks.

Similarity type	Number of isolates	Modularity	Number of communities
Generalized	0	0.47	4
Pearson	0	0.20	4
Cosine	4	0.04	6
Hamming	6	0.00	7

The generalized similarity network has no isolated nodes, the highest modularity, and the smallest number of detected communities. Its community structure partitions the trauma types into the smallest number of well-defined groups of similar size. These groups are compact and homogeneous, and with some insignificant effort can be labeled, as shown in the following table.

Group label	Trauma types
Personal violence	SEXUAL_ASSAULT, SEXUAL_ABUSE, KIDNAP, PSYC_MALTX, WAR, PHYSICAL_ABUSE, PHYSICAL_ASSAULT
Medical traumas	WAR_NOT_US, TRAUMATIC_LOSS, INJURY_ACCIDENT, MEDICAL_TRAUMA
Societal traumas	SCHOOL_VIOLENCE, COMMUNITY_VIOLENCE, EXT_INTERPER_VIOLENCE
Neglect and relocation	IMPAIRED_CAREGIVER, FORCED_DISPLACEMENT, DOMESTIC_VIOLENCE, NATURAL_DISASTER, NEGLECT

This impressive summary completes your analysis of the bipartite network of patients diagnosed with psychological traumas. You started with tabular clinical data pertaining to the trauma cases, and explored four ways of converting the data to a weighted network. You compared the networks, selected the one with the highest modularity, and identified four trauma clusters. (The number of discovered clusters is unrelated to the number of weighted networks.) The results of the study could be even more descriptive if you considered the temporal sequences of traumatic events.

In the Next Part

Your common sense may have suggested that some traumas happen only in a particular order. For example, `SCHOOL_VIOLENCE` does not happen until a child goes to school. The constructed network of traumas cannot reflect the sequencing because it is undirected. The best it can do is to mark two trauma types as similar (if they are). You will work with directed networks in the next part of the book.

Part V

When Order Makes a Difference

Even in the simplest social ego-network, the edges often have directions: their start and end nodes have different semantics. In this part, you will familiarize yourself with directed networks, in particular with directed acyclic graphs, and partitioning them into equivalence classes.

You can't get there from here.

> *Stereotypically attributed to people from Maine*

CHAPTER 17

Directed Networks

Are you a robber or being robbed? Did the Yankees win over the Red Sox or lose to them? Does fish oil provide omega acids or the other way around? Some networks are inherently asymmetric, but we never talk about them—until now.

In this chapter, you will learn how to identify asymmetric relationships between items and build and handle directed networks. In the end, you will be able to check if a directed network is a directed acyclic graph, and if it is, establish a partial order of the nodes by performing a topological sort.

Discover Asymmetric Relationships

A directed network is a network that has at least one directed edge. Naturally, a directed edge is an edge that has a direction: it connects node X to node Y, but not the other way around. Mathematically, the relationship represented by a directed edge is asymmetric.

Many real-world relationships are asymmetric and, ideally, must be modeled as directed networks. Here are some examples of asymmetric or possibly asymmetric relationships:

In social networks:

- Friendship: Alice may believe she is a friend of Bob, but Bob may have a different opinion. In the not-so-rare case when Alice and Bob are mutual friends, you can either model their friendship as an undirected edge or create two anti-parallel directed edges: one from Alice to Bob and the other from Bob to Alice.

- Subordination: if Alice is a subordinate of Bob, then Bob is not a subordinate of Alice.

- Some family relationships, such as parenthood: if Alice is a parent of Bob, then Bob is not a parent of Alice.

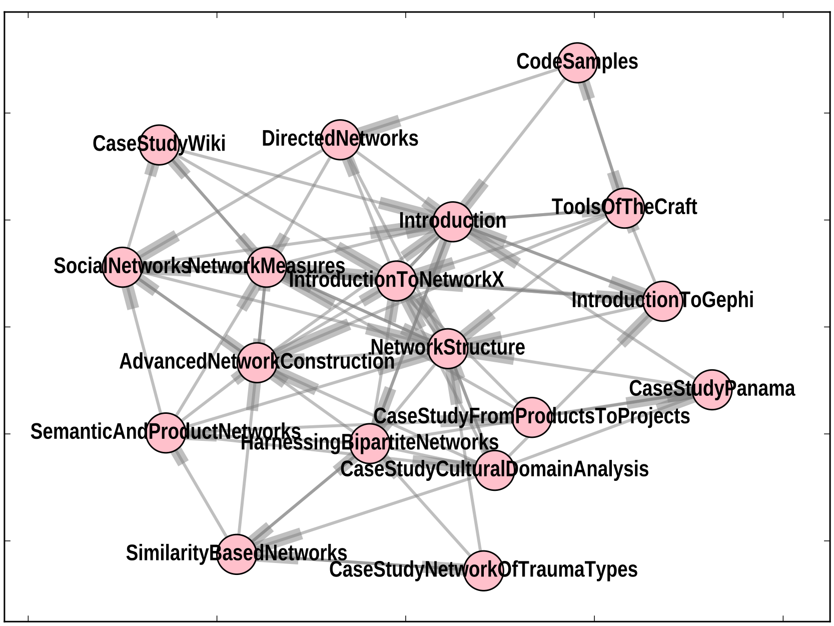
In semantic networks:

- Being a hypernym (a more general word) or a hyponym (a more specific word): *color* is a hypernym of *red* because red is always a color, but the converse is not true.

In other networks:

- Membership: Alice can be a member of an organization, but the organization cannot be a member of Alice.
- Sequencing: if A happens after B, then B does not happen after A.
- WWW links: a link from one web page to another does not imply a reciprocal link.
- Flow: any flow from one node to another (including flows of goods, people, money, and information) is asymmetric and must be modeled as a directed edge. In particular, one-way streets in a transportation network are directed edges.

Even forward and backward references in a book establish an asymmetric relationship between the book units (such as chapters). The figure shows the network of chapters of this book. An edge in the figure connects a chapter to another chapter if there is at least one reference in the former chapter to the latter chapter.



Remember that NetworkX draws rectangles to represent edge arrows. If this unusual notation is difficult to get used to, switch to Gephi for visualization. A chapter whose node has more incident rectangles has more external references and should be an earlier chapter in the book if the goal is to avoid the forward references that some editors consider harmful.

Directed or Signed?

You may feel that the asymmetry of directed networks has something in common with the asymmetry of signed networks (*Signed Networks*, on page 60) and wonder if signed and directed networks model the same aspects. No, they do not. Signed edges (just like all other weighted edges) represent the intensity of the relationship. Directed edges represent its reciprocity. An edge can be (and often is) directed and weighted/signed at the same time. The following table shows how directedness and weight capture different aspects of a simple interpersonal relationship.

	Signed (-)	Unsigned	Signed (+)
Directed	Alice hates Bob (but Bob does not hate Alice).	Alice knows Bob (but Bob does not know Alice).	Alice likes Bob (but Bob does not like Alice).
Undirected	Alice and Bob are foes.	Alice and Bob are acquaintances.	Alice and Bob are friends.

If you represented the same relationship with a "vanilla" unweighted, undirected edge, all the nuances of the Alice and Bob affinity would be irrecoverably lost.

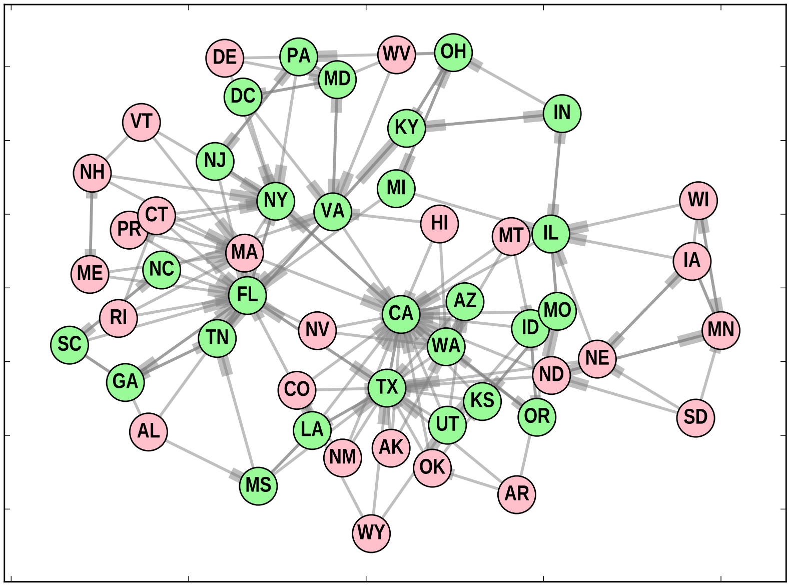
Explore Directed Networks

The directedness of edges dramatically affects almost all network measures and structural elements. Let's have a look at some affected properties. As an example, let's use a directed network of the top three preferred migration destinations for each state in 2015, constructed from the United States Census Bureau State-to-State Migration Flows dataset.[1] You can find the Python code for the network construction in the file migrations.py. The picture on page 200 shows the network sketch (the meaning of the colors will be explained later on page 202).

Degree

Each node in a directed graph G has three degrees: G.in_degree() (the number of incoming incident edges), G.out_degree() (the number of outgoing incident

1. www.census.gov/data/tables/time-series/demo/geographic-mobility/state-to-state-migration.html



edges), and the total G.degree() (the number of all edges). Note that the total degree is the sum of the indegree and outdegree.

The indegree and total degrees of the migration graph designate the most attractive destinations (which are, not surprisingly, sunny California, Florida, and Texas). By construction, all nodes have the same outdegree of 3.

```
sorted(G.in_degree().items(), key=lambda x: x[1], reverse=True)[:3]
sorted(G.out_degree().items(),key=lambda x: x[1], reverse=True)[:3]
sorted(G.degree().items(),    key=lambda x: x[1], reverse=True)[:3]
```

```
[('CA', 21), ('FL', 17), ('TX', 16)]
[('KY', 3), ('MT', 3), ('MS', 3)]
[('CA', 24), ('FL', 20), ('TX', 19)]
```

Neighbors

A node in a directed graph has two types of neighbors: G.successors() (reachable through the outgoing edges) and G.predecessors() (reachable through the incoming edges). The method G.neighbors() is another name of G.successors(). In the migration network, the successors of the final_destination (*CA*) are the preferred destinations of the outgoing migration. The successors are the states from which migrants come to California.

```
final_destination = sorted(G.in_degree().items(), key=lambda x: x[1],
                           reverse=True)[0][0]
coming_from = G.predecessors(final_destination)
```

```
❮ ['LA', 'TX', 'UT', 'ND', 'NY', 'NM', 'NV', 'VA', 'WA', 'ID', 'AK', 'MO',
   'CO', 'OR', 'MT', 'HI', 'KS', 'AZ', 'IL', 'MA', 'OK']
```

```
going_to = G.successors(final_destination)
```

```
❮ ['NY', 'TX', 'AZ']
```

Walks, Trails, and Paths

A walk in a network is *still* any sequence of edges such that the end of one edge is the beginning of another edge (see *Think in Terms of Paths*, on page 88). However, a directed edge has only one end and one beginning, while an undirected edge may begin at any of its incident nodes. Some undirected walks may become broken as a result of this additional restriction (think of encountering a one-way street going in the "wrong" direction).

Centralities and Other Distances

Each node in a directed graph has three degree centralities (G.in_degree_centrality(), G.out_degree_centrality(), and G.degree_centrality()), based on the namesake degrees. The other types of centralities—closeness, betweenness, and eigenvector—are calculated the same way for directed and undirected networks, but the results, in general, differ because of the different neighborhoods and paths. The latter is also true about the center, diameter, radius, eccentricity, and the periphery of a graph (*Networks as Circles*, on page 91).

Components

A directed network has two types of components, as explained in *Split Networks into Connected Components*, on page 126. In a strongly connected component, any member node is reachable from any other member node. (There is a migration flow from any state to any state, perhaps through some intermediate states in the same component.) In a weakly connected component, any member node would be reachable from any other member node if all edges were converted to undirected. (There is a migration flow *either from or to* any state, perhaps through some intermediate states in the same component.)

```
sorted(nx.weakly_connected_components(G),   key=len, reverse=True) # Only one!
sorted(nx.strongly_connected_components(G), key=len, reverse=True)
```

```
❮ [{'ND', 'MD', 'IL', 'CA', 'FL', 'NE', 'IN', 'NH', 'GA', 'ME', 'UT', 'PA',
    'AZ', 'PR', 'VT', 'MT', 'NJ', 'MA', 'WV', 'AK', 'DC', 'MN', 'TX', 'AL',
    'NM', 'MO', 'WI', 'WA', 'OR', 'LA', 'NV', 'IA', 'NC', 'MS', 'CO', 'WY',
    'HI', 'SC', 'TN', 'CT', 'RI', 'DE', 'AR', 'OK', 'NY', 'KY', 'OH', 'KS',
    'MI', 'ID', 'VA', 'SD'}]
```

```
[{'MD', 'IL', 'CA', 'FL', 'IN', 'GA', 'PA', 'UT', 'AZ', 'NJ', 'DC', 'TX',
  'MO', 'WA', 'LA', 'OR', 'NC', 'MS', 'SC', 'TN', 'OH', 'NY', 'KY', 'KS',
  'MI', 'ID', 'VA'}, {'NE', 'WI', 'ND', 'MN', 'IA'}, {'NH', 'ME'}, {'MA'},
  {'VT'}, {'PR'}, {'MT'}, {'AL'}, {'NV'}, {'CO'}, {'WY'}, {'CT'}, {'RI'},
  {'DE'}, {'OK'}, {'AR'}, {'SD'}, {'NM'}, {'WV'}, {'AK'}, {'HI'}]
```

Function nx.condensation(G) calculates the condensation of G. A condensation is an induced directed graph whose nodes represent strongly connected components of G, and edges represent bundles of the original edges, in the same spirit as explained on page 133. All original graph nodes within an induced node of the condensation are definitely reachable from each other. If your goal is to study graph reachability, replacing a strongly connected component with one node does not affect your findings, but makes the problem simpler.

A strongly connected component is called *attracting* if it has no outgoing edges whatsoever. NetworkX offers functions nx.attracting_components(G) and nx.attracting_component_subgraphs(G) to obtain the attracting components. The figure on page 200 shows the nodes in the attracting component in green. Once you move into a “green” state, you will likely stay in the “green” state for good.

```
sorted(nx.attracting_components(G), key=len, reverse=True)
```

```
❮ [{'MD', 'IL', 'CA', 'FL', 'IN', 'GA', 'PA', 'UT', 'AZ', 'NJ', 'DC', 'TX',
  'MO', 'WA', 'LA', 'OR', 'NC', 'MS', 'SC', 'TN', 'OH', 'NY', 'KY', 'KS',
  'MI', 'ID', 'VA'}]
```

Reversal and Flattening

You cannot live your life backward or gather spilled milk, but you can reverse a directed graph with the method G.reverse(). The function returns a copy of the original graph with each edge reversed. The indegrees, outdegrees, successors, and predecessors in the new graph are the outdegrees, indegrees, predecessors, and successors of the original graph, respectively. Both graphs have the same weakly and strongly connected components. If your graph represents consequences for each cause, the reversed one shows all the causes for each consequence.

Finally, NetworkX provides a tool for getting rid of directedness altogether. Method G.to_undirected(reciprocal=False) returns an undirected copy of a directed graph. If the parameter reciprocal is True, then the function connects two nodes with an undirected edge only if they are already connected by a pair of antiparallel directed edges. Otherwise, directed edges are demoted to undirected edges, and any possible resulting pairs of parallel edges are merged.

Apply Topological Sort to Directed Acyclic Graphs

A directed acyclic graph (DAG) is a special type of directed network. As the name suggests, it is an acyclic network—a network that does not contain any cycles (cycles are explained on page 89). Visually, a DAG is a tree, a forest, a star, or a linear graph—for example, the linear graph and the tree in the figure on page 4 are DAGs.

Directed acyclic graphs describe hierarchies—systems in which their components are ranked one above the other according to some property. (A hierarchy is often informally referred to as a "pecking order": who pecks whom?) In a hierarchy, any two components A and B are either unrelated, or A is unambiguously subordinated to B, or B is unambiguously subordinated to A, either directly or indirectly. On the contrary, subordination is ambiguous in directed graphs with cycles. For example, in a two-node ring consisting only of A and B, both nodes can claim that they supervise the other node.

Pecking Order

The edges of a directed acyclic graph often represent a dominance hierarchy or subordination. The source node of an edge is the "boss," and the target node is a "subordinate." Incidentally, dominance in chickens is asserted by pecking. The "top" chicken pecks a more inferior chicken, which, in turn, pecks an even more inferior chicken, all the way down to the "bottom" chicken. Pecking order effectively executes a topological sort and leads to social stratification.

All NetworkX functions and techniques for directed networks naturally work for DAGs, but several functions are intended solely for DAGs. Function `nx.is_directed_acyclic_graph(G)` checks if G is a DAG or not. Function `nx.transitive_closure(G)` calculates a transitive closure T of G: a graph that has the same nodes as G such that two nodes in T are adjacent if and only if there is a path between the two nodes in G. Think of a transitive closure as a graph of all possible subordination relationships, both direct and indirect.

You can serialize a DAG and arrange all nodes in a linear order, so that the next node may be a subordinate of the previous node, but the previous node is never a subordinate of the next node. The result of the serialization is a ranking of all nodes, with the source nodes at the beginning and target nodes at the end. This operation is called topological sort. You can sort a DAG in many different ways, resulting in different "pecking" rankings. Function `nx.topological_sort(G)` returns one randomly chosen ranking as a list of node labels.

```
nx.topological_sort(G)
```

```
❮ ['NM', 'DC', 'AR', 'HI', 'MI', 'SC', 'PR', 'MS', 'TN', 'CT', 'AL', 'ME',
   'OR', 'VT', 'UT', 'DE', 'NC', 'NH', 'WY', 'CO', 'OK', 'IN', 'AK', 'WA',
   'SD', 'AZ', 'KS', 'MO', 'RI', 'MA', 'LA', 'TX', 'MD', 'NE', 'IA', 'NV',
   'MT', 'ID', 'WV', 'VA', 'KY', 'OH', 'PA', 'NJ', 'ND', 'MN', 'WI', 'IL',
   'CA', 'GA', 'FL', 'NY']
```

A topological sort order is not too useful because it focuses on what is impossible rather than on what is definite. You can tell from the order that New Mexico (NM) is not one of the top five destinations for the residents of New York (NY), but you cannot claim that New York is one of the top five destinations for the inhabitants of New Mexico.

Master "toposort"

Directed network analysis has an unexpected connection to creative writing and computer game development.

> This section uses Pandas, community, toposort.

Game developers and creative writers are often in need of a collection of adjectives that characterize a particular property and range from "very bad" to "very good." Directed network analysis (via the module toposort) makes it possible to design such a scale in any natural language.

Obtain and Extract Survey Data

You can start this mini case study by defining a list of candidate adjectives; in our case, the list consists of thirty-four words: "alpha plus," "average," "bad," "crappy," "disgusting," "excellent," "exciting," "f*cking good," "fantastic," "filthy," "first-class," "good," "great," "horrible," "lousy," "magical," "mediocre," "pathetic," "nice," "none of a," "normal," "not bad," "phenomenal," "premium," "repugnant," "shitty," "so-so," "solid," "strong," "superb," "superior," "unfit," "weak," and "worthless."

"Babushka," "Sputnik," "Balalaika"…

The original data for this case study was collected by me in 2016 in the Russian language and later translated into English. You may argue that the perception and interpretation of qualitative adjectives in the two languages differ, and you are probably right. However, the goal of the project is to introduce and illustrate the technique, rather than produce an actual highly reliable line of adjectives.

Post the list as a survey to Qualtrics,[2] SurveyMonkey,[3] or your other favorite survey-taking site. You have to design the questionnaire in such a way that the takers either rank all words in the order from the “best” to the “worst” or assign a numerical measure of “goodness” to each word. The survey design is outside the scope of this book, but keep in mind that asking survey takers to arrange thirty-four words on a cellphone screen may be more than an average person is ready to commit to.

Once you collect enough samples, download the results as a CSV file (say, Adjectives_by_the_rank.csv). Depending on the surveying site, the file may need a lot of cleanup before becoming useful. The following code fragment imports a CSV file produced by Qualtrics, extracts the thirty-four columns that correspond to the word ranks, and removes the survey question from the column names.

adjectives.py

```
ranks = pd.read_csv("Adjectives_by_the_rank.csv",
                    header=1).set_index("ResponseID").fillna(0)
Q1 = "Rank the words from the most positive to the most negative-"
ranks = ranks.loc[:, ranks.columns.str.startswith(Q1)].astype(int)
ranks.columns = ranks.columns.str.replace(Q1, "")
```

Let’s now build a network of words. Each column of the DataFrame ranks represents the ranks of a word from each participant. Connect the word i to another word j with a directed edge if the participants agree, to some extent, that i is “better” than j. The definition of what constitutes the agreement may be stringent (by consensus), weak (when at least two participants agree), or somewhere in the middle (say, at least 115 of 158 participants agree). The consensus-based network would have very few, if any, edges. The network based on the weak criterion may have too many edges and contain cycles. We want to construct a network that is dense but still has no cycles, because if it is not a DAG, then it cannot be topologically sorted.

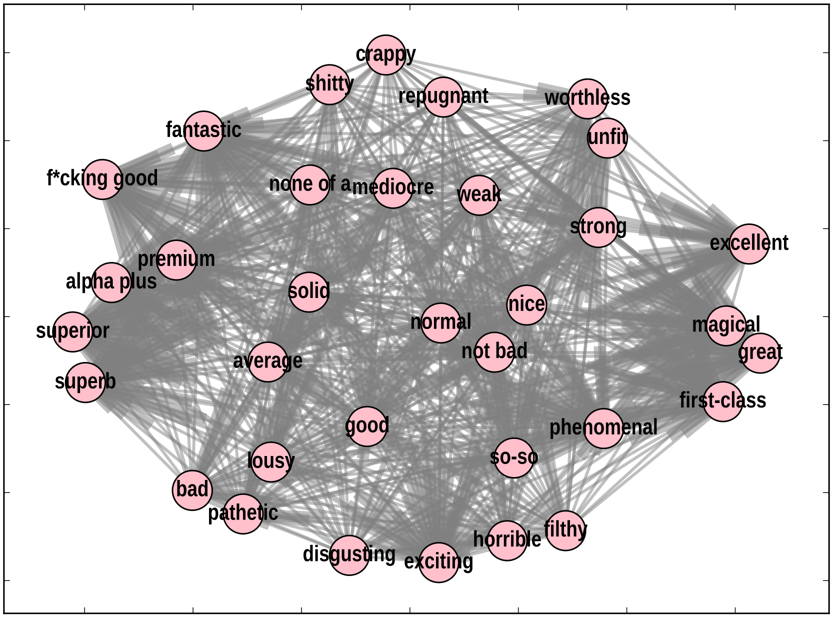
adjectives.py

```
dominance = pd.DataFrame([[(ranks[j] > ranks[i]).sum()
                           for i in ranks] for j in ranks],
                         columns=ranks.columns, index=ranks.columns)

QUORUM = 115
edges = sorted(dominance[dominance >= QUORUM].stack().index.tolist())
G = nx.DiGraph(edges)
```

2. www.qualtrics.com
3. www.surveymonkey.com

The resulting network G has thirty-four nodes (one node per qualitative word) and 497 edges. It consists of one dense weakly connected component shown in the following figure.



Amazingly, the graph is a DAG.

```
nx.is_directed_acyclic_graph(G)
```

```
❮ True
```

Unfortunately, it looks incomprehensible.

Execute Topological Sort

You can try to bring some order by topologically sorting the network (*Apply Topological Sort to Directed Acyclic Graphs, on page 203*):

adjectives.py

```
# Sort in the reverse order
print(nx.topological_sort(G)[::-1])
```

```
❮ ['exciting', 'fantastic', 'phenomenal', 'superior', 'first-class', 'magical',
  'f*cking good', 'superb', 'premium', 'alpha plus', 'great', 'excellent',
  'strong', 'good', 'solid', 'not bad', 'nice', 'normal', 'average',
  'mediocre', 'none of a', 'so-so', 'weak', 'bad', 'unfit', 'worthless',
  'pathetic', 'lousy', 'shitty', 'horrible', 'filthy', 'repugnant',
  'disgusting', 'crappy']
```

The output of the function is unbelievably realistic. "Fantastic" is undeniably better than "great," which is better than "lousy," which is better than "disgusting." The only problem with the function nx.topological_sort(G) is that it returns only one possible topological sort order. It forcefully ranks the nodes that are otherwise topologically equivalent, adding unnecessary constraints to the way the node labels can be used. There is no way to obtain another order with nx.topological_sort().

The module toposort[4] provides the function toposort.toposort(edge_dict) that does not have the limitations of the function nx.topological_sort(). This function returns a generator of sets of topologically equivalent nodes. A node in a set does not dominate and is not dominated by any node in the same set. Game developers and creative writers, the prospective users of the word sets, would treat all words in one set as having the same sentiment (but not necessarily the same valence).

The function toposort.toposort(edge_dict) is not integrated with NetworkX. Before using the function, transform a NetworkX edge list G.edges() into a dictionary where nodes are keys, and sets of their neighbors are values.

adjectives.py

```
edge_dict = {n1: set(ns) for n1, ns in nx.to_dict_of_lists(G).items()}
topo_order = list(toposort.toposort(edge_dict))
print(topo_order)
```

```
❮ [{'phenomenal', 'exciting', 'fantastic'}, {'f*cking good', 'magical',
   'first-class', 'great', 'superb', 'superior'}, {'alpha plus', 'excellent',
   'premium'}, {'solid', 'strong', 'good'}, {'normal', 'nice', 'not bad'},
   {'average'}, {'none of a', 'mediocre', 'so-so'}, {'weak'}, {'worthless',
   'unfit', 'pathetic', 'bad'}, {'lousy'}, {'shitty', 'filthy', 'horrible',
   'crappy'}, {'repugnant', 'disgusting'}]
```

The new output is "phenomenal," "exciting," and "fantastic." It consists of twelve equivalence classes of word, each class being "worse" than the predecessor and "better" than the successor. Despite being based only on 158 responses, the result does not look unexpected. The toposort algorithm is even "smart" enough to put the unappetizing adjectives in the second set from the end together.

In this chapter, you learned how to identify, capture, and explore (with topological sort) any asymmetric relationships between network nodes. Incidentally, this chapter concludes the main body of the book. Whether or not you were a seasoned complex network analyst and Python programmer at the beginning

4. pypi.python.org/pypi/toposort

of the book, now you are. You will never see the world around you the same way, because *when all you know is CNA, everything looks like a network.*

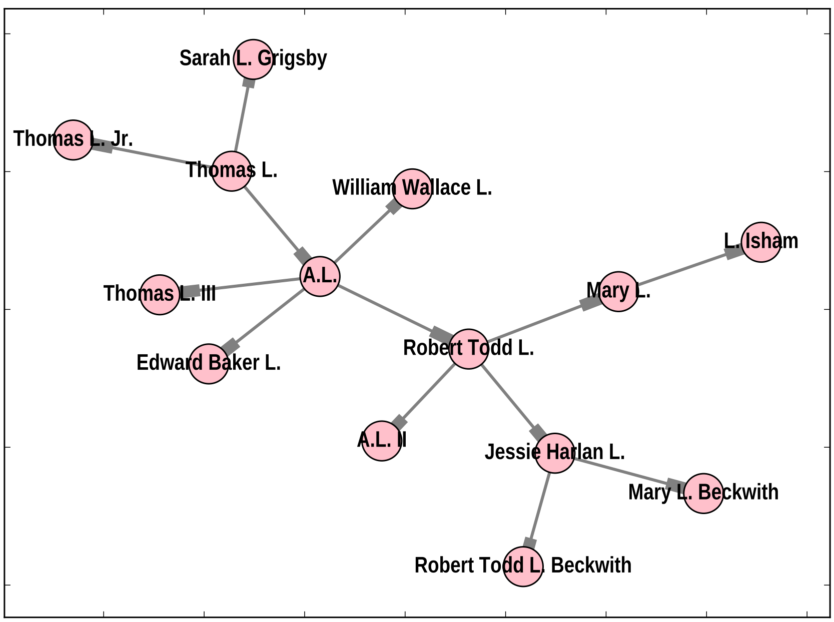
In the Appendix

Just like almost everything, NetworkX evolves. The second major version of the library has been released as this manuscript was in preparation. You will read about the new NetworkX 2.0 in Appendix 2, *NetworkX 2.0*, on page 213.

APPENDIX 1

Network Construction, Five Ways

This appendix compares the ways of constructing the Lincoln family tree network on page 4 (shown in the following figure) in pure Python and using the four toolkits from Chapter 2, *Surveying the Tools of the Craft*, on page 11.



Pure Python

The most natural way to describe a network in pure Python is to represent each edge as a tuple or list of two nodes and collect all edge tuples or list in another list—the edge list.

make-network-type-figures.py

```
lincoln_list = [
    ("A.L.", "Edward Baker L."), ("A.L.", "Robert Todd L."),
    ("A.L.", "William Wallace L."), ("A.L.", "Thomas L. III"),
    ("Jessie Harlan L.", "Mary L. Beckwith"),
    ("Jessie Harlan L.", "Robert Todd L. Beckwith"),
    ("Mary L.", "L. Isham"), ("Robert Todd L.", "A.L. II"),
    ("Robert Todd L.", "Jessie Harlan L."),
    ("Robert Todd L.", "Mary L."), ("Thomas L.", "A.L."),
    ("Thomas L.", "Sarah L. Grigsby"), ("Thomas L.", "Thomas L. Jr."),
]
```

The previous example has at least three major issues:

- Isolated nodes (nodes without edges) cannot be represented because they are not incident to any edges (correctness issue).
- Lists have linear search time (performance issue).
- Node labels are replicated for each incident edge (memory footprint issue).

You could mitigate the first two issues by representing the network as a dictionary, where the keys are node labels, and the values are sets of the adjacent nodes:

```
lincoln_dict = {
   "A.L.": {"Thomas L. III", "Edward Baker L.", "William Wallace L.",
            "Robert Todd L."},
  "Jessie Harlan L.": {"Robert Todd L. Beckwith", "Mary L. Beckwith"},
  "Mary L.": {"L. Isham"},
  "Robert Todd L.": {"Mary L.", "A.L. II", "Jessie Harlan L."},
  "Thomas L.": {"Sarah L. Grigsby", "A.L.", "Thomas L. Jr."},
  "George W.": set()},
```

Note how we incorporated George W[ashington] into the network, despite his not having incident edges yet. (This network representation in used in *Apply Topological Sort to Directed Acyclic Graphs*, on page 203.) The new approach, however, has its own problems:

- Some nodes that have only incoming links (such as "Thomas L. III") are now on the second level of the hierarchy and hard to find.
- You cannot have parallel edges that connect the same two nodes more than once.
- Node labels are still replicated for each incoming incident edge.

iGraph

Here's how iGraph deals with the Lincoln graph. The edge list contains numerical node identifiers rather than labels, but you can add the labels later as node attributes.

```
import igraph
edges = [(1, 6), (1, 7), (1, 5), (1, 12), (2, 4), (2, 9), (3, 13), (7, 0),
         (7, 2), (7, 3), (8, 1), (8, 10), (8, 11)]
labels = [
  "A.L. II", "A.L.", "Thomas L.", "Jessie Harlan L.", "Mary L. Beckwith",
  "Sarah L. Grigsby", "Edward Baker L.", "Mary L.", "William Wallace L.",
  "Robert Todd L.", "Robert Todd L. Beckwith", "Thomas L. Jr.",
  "Thomas L. III", "L. Isham", "George W."]
G = igraph.Graph(edges, directed=True)
G.add_vertex(14)
G.vs["name"] = labels
print(G)
```

```
IGRAPH DN-- 15 13 --
+ attr: name (v)
+ edges (vertex names):
A.L.->Edward Baker L., A.L.->Mary L., A.L.->Sarah
L. Grigsby, A.L.->Thomas L. III, Thomas L.->Mary L. Beckwith,
Thomas L.->Robert Todd L., Jessie Harlan L.->L. Isham, Mary
L.->A.L. II, Mary L.->Thomas L., Mary L.->Jessie
Harlan L., William Wallace L.->A.L., William Wallace L.->Robert Todd L.
Beckwith, William Wallace L.->Thomas L. Jr.
```

Alas, the unconnected nodes are not shown on the printout! Poor George W.

graph-tool

The graph-tool version of the Lincoln graph starts with the same edge list and a list of labels. All vertices are added at once, followed by all edges. graph-tool treats labels as vertex properties.

```
import graph_tool
edges = [(1, 6), (1, 7), (1, 5), (1, 12), (2, 4), (2, 9), (3, 13), (7, 0),
         (7, 2), (7, 3), (8, 1), (8, 10), (8, 11)]
labels = [
  "A.L. II", "A.L.", "Thomas L.", "Jessie Harlan L.", "Mary L. Beckwith",
  "Sarah L. Grigsby", "Edward Baker L.", "Mary L.", "William Wallace L.",
  "Robert Todd L.", "Robert Todd L. Beckwith", "Thomas L. Jr.",
  "Thomas L. III", "L. Isham", "George W."]

G = graph_tool.Graph() # Directed by default
nodes = G.add_vertex(len(labels))
G.add_edge_list(edges)

# Add node labels ("vertex properties")
names = G.new_vertex_property("string")
for node,label in zip(nodes, labels):
    names[node] = label
```

NetworkX

The striking difference between the NetworkX code and the other previously shown code fragments is the ability to add named edges and nodes directly to the graph, which is a natural way of building a real-world complex network.

make-network-type-figures.py

```
lincoln_list = [
    ("A.L.", "Edward Baker L."), ("A.L.", "Robert Todd L."),
    ("A.L.", "William Wallace L."), ("A.L.", "Thomas L. III"),
    ("Jessie Harlan L.", "Mary L. Beckwith"),
    ("Jessie Harlan L.", "Robert Todd L. Beckwith"),
    ("Mary L.", "L. Isham"), ("Robert Todd L.", "A.L. II"),
    ("Robert Todd L.", "Jessie Harlan L."),
    ("Robert Todd L.", "Mary L."), ("Thomas L.", "A.L."),
    ("Thomas L.", "Sarah L. Grigsby"), ("Thomas L.", "Thomas L. Jr."),
]

import networkx as nx
G = nx.DiGraph(lincoln_list) # Directed!
G.add_node("George W.")
```

NetworKit

You can convert a previously constructed NetworkX graph G into a NetworKit graph nkG with a call to one function.

```
import networkit.nxadapter
G = ... # Construct a NetworkX graph
nkG = networkit.nxadapter.nx2nk(G)
```

We cannot be certain of being right about the future; but we can be almost certain of being wrong about the future, if we are wrong about the past.

➤ *Gilbert Keith Chesterton, English writer, poet, philosopher, dramatist, journalist, orator, lay theologian, biographer, and literary and art critic*

APPENDIX 2

NetworkX 2.0

A new version of NetworkX—2.0—was released in September 2017. The new version is only partially compatible with the stable version used in the book. The book describes over one hundred NetworkX functions. Some of them have been affected by the transition.

Surely, the new version does not instantly make version 1.11 obsolete. 2.0 will be gradually phased in, while 1.11 will be phased out. In this appendix, you will read about the most striking differences between the old and the new versions, which will help you to adjust your CNA scripts to work with the most recent version of NetworkX. A complete migration guide is available online.[1]

From Containers to Views

- Function nx.subgraph() and methods G.subgraph(), G.neighbors(), G.reverse(), G.to_directed(), and G.to_undirected(), to name a few, return View objects instead of graphs. A view refers to the underlying graph. Any node, edge, or attribute change in the underlying graph affects all of the associated views.

- New attributes G.nodes and G.edges contain dict()-like NodeView and EdgeView objects, respectively. Methods G.nodes() and G.edges() return the namesake views, too.

- New attributes G.degree, G.in_degree, and G.out_degree contain dict()-like DegreeView objects.

Usage Change

- Function bipartite.sets() raises an AmbiguousSolution exception if the input bipartite graph is disconnected, because it is not possible to separate bipartite sets in a disconnected graph unambiguously.

1. networkx.github.io/documentation/stable/release/migration_guide_from_1.x_to_2.0.html

- The order of parameters changed for the functions nx.set_edge_attributes() and nx.set_node_attributes(). The list of attribute values is now the second parameter, and the attribute name is the third parameter.
- Method G.selfloop() became function nx.selfloop().

Deprecation

- Function nx.blockmodel() deprecated in favor of nx.quotient_graph() with relabel=True.
- Functions nx.from_pandas_dataframe() and nx.to_pandas_dataframe() deprecated in favor of nx.to_pandas_adjacency(), nx.from_pandas_adjacency(), nx.to_pandas_edgelist(), and nx.from_pandas_edgelist().

Bibliography

[BA99] A-L Barabási and R Albert. Emergence of Scaling in Random Networks. *Science*. 286[5439]:509–512, 1999.

[Bar03] A-L Barabási. *Linked*. Plume, New York, NY, 2003.

[Bar54] J Barnes. Class and Committees in a Norwegian Island Parish. *Human Relations*. [7]:39–58, 1954.

[BE05] U Brandis and T Erlebach. *Network Analysis: Methodological Foundations*. Springer, New York, NY, 2005.

[BEJ13] SP Borgatti, MG Everett, and JC Johnson. *Analyzing Social Networks*. SAGE Publications, Los Angeles, CA, 2013.

[BGL11] A-L Barabási, N Gulbahce, and J Loscalzo. Network Medicine: a Network-Based Approach to Human Disease. *Nature Reviews Genetics*. 12:56–58, 2011, January.

[BGLL08] VD Blondel, J-L Guillaume, R Lambiotte, and E Lefebvre. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment*. [10]:P10008, 2008.

[BKG11] M Buhrmester, T Kwang, and SD Gosling. Amazon's Mechanical Turk: A New Source of Inexpensive, Yet High-Quality, Data?. *Perspectives on Psychological Science*. 1[6]:3–5, 2011.

[BKL09] S Bird, E Klein, and E Loper. *Natural Language Processing with Python*. O'Reilly & Associates, Inc., Sebastopol, CA, 2009.

[BWR17] HR Bernard, AY Wutich, and GW Ryan. *Analyzing Qualitative Data. Systematic Approaches*. SAGE Publications, Los Angeles, CA, 2017.

[CF09] NA Christakis and JH Fowler. *Connected*. Back Bay Books, New York, NY, 2009.

[DGG41] A Davis, BB Gardner, and MR Gardner. *Deep South*. University of Chicago Press, Chicago, IL, 1941.

[Dun98] R Dunbar. *Grooming, Gossip, and the Evolution of Language*. Harvard University Press, Boston, MA, 1998.

[EG12] T Eliassi-Rad and M Gupte. Measuring Tie Strength in Implicit Social Network. *Proc. 4th ACM International Conference on Web Science*. 2012, June.

[EK10] D Easley and J Kleinberg. *Networks, Crowds, and Markets. Reasoning about a Highly Connected World*. Cambridge University Press, Cambridge, United Kingdom, 2010.

[Gra73] M Granovetter. The Strength of Weak Ties. *American Journal of Sociology*. 78[6]:1360–1380, 1973, May.

[Gra83] M Granovetter. The Strength of Weak Ties: a Network Theory Revisited. *Sociological Theory*. 1:201–233, 1983.

[GZ17] S Gantman and D Zinoviev. Conceptual Structure of Fraud Research and Its Dynamics. *Proc. AMCIS2017 on Accounting Information Systems*. [4], 2017, August.

[HSZL17] HB Hodgdon, MK Suvak, D Zinoviev, R Liebman, and EC Briggs. Network Analysis of Exposure to Trauma and Adverse Events in a Clinical Sample of Children and Adolescents. *Under review*. 2017.

[Jac08] M Jackson. *Social and Economic Networks*. Princeton University Press, Princeton, NJ, 2008.

[KJFC15] J Krause, R James, D Franks, and D Croft. *Animal Social Networks*. Oxford University Press, New York, NY, 2015.

[Kov10] B Kovacs. A Generalized Model of Relational Similarity. *Social Networks*. 32[3]:197–211, 2010.

[KY08] D Knoke and S Yang. *Social Network Analysis*. SAGE Publications, Los Angeles, CA, Second edition, 2008.

[Lew09] TG Lewis. *Network Science. Theory and Applications*. John Wiley & Sons, New York, NY, 2009.

[Mor34] JL Moreno. *Who Shall Survive?*. Beacon House, New York, NY, 1934.

[New06] MEJ Newman. Modularity and Community Structure in Networks. *Proc. National Academy of Sciences of the United States of America*. 103[23]:8577–8696, 2006.

[New10] MEJ Newman. *Networks: An Introduction*. Oxford University Press, New York, NY, 2010.

[NMB11] W de Nooy, A Mrvar, and V Batagelj. *Exploratory Social Network Analysis with Pajek (Structural Analysis in the Social Sciences)*. Cambridge University Press, Cambridge, United Kingdom, Expanded edition, 2011.

[PBMW99] L Page, S Brin, R Motwani, and T Winograd. The PageRank Citation Ranking: Bringing Order to the Web. *Stanford InfoLab*. 1999.

[PDFV05] G Palla, I Derenyi, I Farkas, and T Vicsek. Uncovering the Overlapping Community Structure of Complex Networks in Nature and Society. *Nature*. 435:814–818, 2005.

[RW99] R Read and R Wilson. *An Atlas of Graphs*. Oxford University Press, New York, NY, 1999.

[Sco00] J Scott. *Social Network Analysis: A Handbook*. SAGE Publications, Los Angeles, CA, Second edition, 2000.

[Str01] S Strogatz. Exploring Complex Networks. *Nature*. 410[6825]:268–276, 2001.

[Wat03] D Watts. *Six Degrees. The Science of a Connected Age*. W. W. Norton & Company, New York, NY, 2003.

[Zin16] D Zinoviev. *Data Science Essentials in Python*. The Pragmatic Bookshelf, Raleigh, NC, 2016.

[ZLZ15] D Zinoviev, K Li, and Z Zhu. Building Mini-Categories in Product Networks. *Complex Networks VI*. 597:179–189, 2015.

Index

SYMBOLS

[] (selection operator), 23

A

A4 page format, 40
Abraham Lincoln timeline
- converting adjacency matrix, 70–75
- with graph-tool, 211
- with iGraph, 210
- moving data with edge lists and node dictionaries, 76–77
- with NetworKit, 212
- with networkx, 212
- with pure Python, 209
- slicing, 80
- visualizations, 4

actors, *see* nodes
acyclic graphs, directed, 203–208
Adamic, Lada, 55
add_edge(), 19, 23
add_edges_from(), 19, 23, 71
add_node(), 19, 23
add_nodes_from(), 19, 23
add_weighted_edges_from(), 24
adjacency
- bipartite networks, 180–181, 187
- clique communities, 134
- creating networks from adjacency matrices, 69–75
- defined, 17
- importing and exporting adjacency lists, 30

adjectives for game developers example, 204–208
algorithm.bipartite module, 177
Alice, 62
all_neighbors(), 85
alter nodes, 54–57, 84–86
Amazon's Mechanical Turk, 139
Anaconda, xv, 136
angular distance and cosine distance, 172
Animal Social Networks, 53
anti-communities, 136
anti-parallel directed edges, 197
assortative mixing, 97, 105
assortativity
- coefficient, 100, 105
- cosmetics case study, 156
- defined, 97
- estimating uniformity, 97–100
- Panama Papers case study, 105–107
- in social network analysis, 6

asymmetry
- directed graphs, 197–199
- examples, 197
- family trees, 3
- signed networks, 199
- timelines, 3

attracting components, 202
attracting_components(), 202
attracting_components_subgraphs(), 202
attribute_assortativity_coefficient(), 100, 105
attribute_mixing_matrix(), 99, 105
attributes, *see also* similarity
- Abraham Lincoln timeline, 210
- adding, 23–25
- assortativity, 97–100, 105–107
- binarizing, 166
- changing, 24
- co-occurrence, 115
- contractions, 45
- defined, 2
- editing appearance, 31
- in graph-tool, 15
- Hamming distance, 168
- handling with Pandas, 74
- Manhattan distance, 169
- networkx 2.0, 213
- normalizing for Manhattan distance, 170
- processing selectively, 109
- removing, 24
- selecting incident edges, 23
- storage in networkx, 20

authorities, 35, 95–96
average_clustering(), 88
average_degree_connectivity(), 98

B

balance theory, 57
balanced_tree(), 78
barabasi_albert_graph(), 78
Barabási, Albert-László, 129

Barabási–Albert graphs
about, 65
generating synthetic networks, 78
Panama Papers case study, 106
Barnes, John, 6
BeautifulSoup, 154
best-partition(), 191
betweenness centrality
bipartite networks, 178
defined, 93
directed graphs, 201
ego networks, 57
measuring, 35, 96, 201
Othello semantic network, 119
product networks, 122
social networks, 57
betweenness_centrality(), 96, 178
BFS (breadth-first search), 43
bi-adjacency matrix, 180, 187
binomial graphs, *see* Erdös–Rényi graphs
biological networks, examples, 5
bipartite networks, 175–183
anti-communities, 136
checking for, 177, 187
defined, 175
disadvantages, 176
examples, 176
functions, 177–178
networkx 2.0, 213
projecting, 178–183, 190
sketching by hand example, 7
trauma types case study, 185–192
bipartite.sets(), 177, 213
blockmodel(), 138
blockmodeling
core-peripheral analysis, 129
cosmetics case study, 157
defined, 138
deprecation in networkx 2.0, 214
with graph-tool, 14
naming extracted blocks, 139
performing, 138
term, 138
Bob, 62
bottom nodes, bipartite networks, 178–183
branching factor, 78
Brandes, Ulrik, 95
breadth-first search (BFS), 43
bridges, 66, 94
broadcasting and communication networks, 62
brokerage, 57, 93
Building Mini-Categories in Product Networks, 122

C

C
graph-tool, 13, 16
iGraph support, 12, 16
limitations for CNA, xiii
C++, 13, 16
caching, data for cultural domain analysis, 143
case, lowering, 145
case studies, *see also* Wikipedia pages case study
cosmetic products, 153–160
cultural domain analysis, 141–151
Panama Papers, 101–111, 176
trauma types network, 185–192
CDA, *see* cultural domain analysis
center, measuring, 91, 201
center() method, 91
centrality, *see also* betweenness centrality; eccentricity; eigenvector centrality
closeness centrality, 35, 93, 96, 178, 201
degree centrality, 35, 92, 201
directed graphs, 201
harmonic centrality, 93, 96
HITS (Hyperlink-Induced Topic Search), 35, 95–96
measuring, 32, 35, 90, 92–97, 201
measuring correlation of, 96
Othello semantic network, 119
PageRank, 35, 94–96
social network examples, 57
chain.from_iterable(), 155
chat sessions, recording, 63
checkerboard networks, *see* grids
Chuck, 62
circles, networks as, 91
circular layout, 26–28
circular_layout(), 26
cityblock(), 170
classic networks, *see also* simple networks
generating synthetic networks, 78
types, 2–4, 63
clear(), 20
clique percolation, 134
clique-node, 133
cliques
bipartite networks, 178
clique-node, 133
defined, 131
extracting, 131–134
maximal, 132–133, 138
maximum, 132
percolation, 134
recognizing clique communities, 134–135
cliques, social, 66
closeness centrality
bipartite networks, 178
defined, 93
directed graphs, 201
measuring, 35, 96, 201
closeness_centrality(), 96, 178
clustering, *see* clustering coefficients; community detection; modularity
clustering coefficients
bipartite networks, 178
clustering as term, 88
clusters as term, 134
measuring, 32, 35, 87
clustering(), 87, 178
CNA, *see* complex network analysis
co-occurrence
component analysis, 128
cultural domain analysis case study, 147
defined, 115
defining for analysis, 119

product networks, 120–123
semantic networks, 116–120
code for this book
cosmetics case study data, 154
cultural domain analysis case study data, 141
dzcnapy_plotlib module, 27
migration distribution data, 199
modules needed, xv
notation, xvi
online files, xvii
trauma types case study data, 186
cognitive balance, *see* homophily
coincidence, *see* co-occurrence
collections, cosmetics case study, 153–160
communication networks, understanding, 61
communities, *see also* cliques
anti-communities, 136
blockmodeling, 138
community detection, 12, 14, 16
cosmetics case study, 157
cultural domain analysis case study, 148–150
defined, 35
modularity-based, 136–138, 191
partitioning into, 35, 88, 148–150, 157, 191
social network examples, 57
term, 134
trauma types case study, 191
community
cultural domain analysis case study, 148–150
managing modularity-based communities, 136
trauma types case study, 191
version, xv
community detection
graph-tool, 14, 16
iGraph, 12, 16
networkx, 12, 16
complements in product networks, defined, 120
complete graphs
about, 64
clustering coefficient, 87
generating, 78
complete subgraphs, *see* cliques
complete_graph(), 78
complex contagion, 57
complex network analysis, *see also* case studies; networks
bipartite networks, 178
cautions about building own modules, 11
defined, xiii, 1
with Gephi, 32, 34–37
history, 1, 5
as iterative process, 17
use in medical and public health, 185
complex networks, *see also* cliques; networks; similarity
defined, 4
major classes, 5
separating cores, shells, coronas, and crusts, 129–131
sketching by hand, 6–8
splitting into connected components, 126–128
structural elements, 125–139
synthetic networks, 78
components
attracting components, 202
condensation, 202
cosmetics case study, 155–160
detection, 90
directed acyclic graphs, 203
directed graphs, 201
filtering, 104
giant connected component (GCC), 125, 128–129, 155
measuring connectedness with Gephi, 32, 35
naming, 157–160
Panama Papers case study, 104
reversal, 202
splitting networks into, 126–128
components, connected
compared to cliques, 131
cosmetics case study, 155
defined, 126
filtering, 104
separating cores, shells, coronas, and crusts, 129–131
splitting networks into, 126–128
Conceptual Structure of Fraud Research and Its Dynamics, 116
condensation, 202
condensation() method, 202
connected_component_subgraphs(), 104, 128
connected_components(), 127, 155
connected_watts_strogatz_graph(), 78
connectedness, *see also* assortativity
measuring, 32, 35
Panama Papers case study, 104
splitting networks into connected components, 126–128
consequences, reversal, 202
contagion, xv, 57
Continuum Analytics, xv
contracted_nodes(), 45
contractions, 45
core
defined, 129–130
main, 130
separating, 129–130
core nodes, collecting, 47
core-peripheral analysis, 129
corona
defined, 130
separating, 129–130
corr(), 96, 174
correlation, *see* Pearson correlation
cosine similarity, 171–173, 187–192
cosine(), 172
cosmetic products case study, 153–160

Counter(), 158
coupling, product networks, 122
crust
  defined, 130
  separating, 129–130
CSV files
  adjectives for game developers example, 205
  cosmetics case study, 153–160
  dictionary reader, 103
  food and nutrient example, 21
  Panama Papers case study conversion, 101–111
  trauma types case study, 186
cultural domain analysis
  case study, 141–151
  defined, 141
cultural networks, examples, 5
cycle networks, *see* rings
cycle_graph(), 78
cycles, trails as, 89

D

DAGs, *see* directed acyclic graphs
data alignment, 146, 165
Data Laboratory tab in Gephi, 32–33
Data Science Essentials in Python, 73
DataFrame
  converting adjacency matrices, 73–75
  cultural domain analysis case study, 144, 146, 150
  data alignment, 146, 165
  defined, 73
  importing node attributes, 74
  naming communities helper, 150
  networkx 2.0, 214
  parsing Panama Papers CSV file, 108–111
  parsing irregular texts, 145
  Pearson correlation, 174, 180
  similarity, 164
  as term vector model, 146
Davis Southern women synthetic network
  about, 65
  bipartite networks, 176
  generating, 79, 164
  simplicity in, 164–166
Davis, Allison, 164
davis_southern_women_graph(), 79, 164
degree, *see also* degree centrality; indegree; outdegree
  assortativity, 98, 106
  bipartite networks, 176, 178
  defined, 35
  directed graphs, 199
  giant connected component (GCC), 129
  induced nodes, 159
  networkx 2.0, 213
  Panama Papers case study, 105–107
  in social network analysis, 6
  social network examples, 57
  Wikipedia pages case study, 46
degree assortativity coefficient, 106
degree attribute, 213
degree centrality
  bipartite networks, 178
  defined, 92
  measuring, 35, 92, 96, 201
degree() method, 47, 199
degree(induced) method, 159
degree_assortativity_coefficient(), 105
degree_centrality(), 92, 96, 178, 201
degrees(), 178
DegreeView, 213
deleting
  all nodes and edges while keeping shell, 20
  attributes, 24
  duplicates, 45, 110
  isolates, 126
  nodes and edges with Gephi, 31
  nodes and edges with networkx, 19, 45, 110
  nodes from ego networks, 55
  self-loops, 22, 45
density
  bipartite networks, 178, 189
  converting sparse matrix to dense, 76
  defined, 84
  measuring, 35, 84
density() method, 84, 178
describe_cluster(), 150
deserialization, with pickle, 143, *see also* serialization
diameter
  defined, 91
  directed graphs, 201
  measuring, 35, 91, 201
diameter() method, 91
dichotomization, 166
dictionaries
  attributes, 23
  converting weight to, 147
  CSV dictionary reader, 103
  of dictionaries, 77
  of lists, 77
  measuring length, 21
  moving data with node, 76–77
  node and edge storage, 20
  relabeling nodes, 22
  to mitigate issues with edge lists, 210
digital humanities, 118
DiGraph(), 18
digraphs, *see* directed graphs
directed acyclic graphs, 203–208
directed edges
  about, 18
  anti-parallel, 197
  asymmetric relationships, 197–199
  defined, 197
  networkx visualization, xvi
  social networks, 53
directed graphs, 197–208
  asymmetric relationships, 197–199
  clique strength, 131
  clustering coefficient, 87
  condensation, 202
  connected components, 126

converting to undirected, 18, 128, 202
creating, 18
defined, 18, 197
degree, 199
density, 84
directed acyclic graphs, 203–208
directed multigraphs, 19
eccentricity, 91, 201
flattening, 202
measurement, 199–208
networkx 2.0, 213
PageRank, 95
reversal, 202, 213
vs. signed networks, 199
social networks, 53
visualizations with Gephi, 199
weight, 199
Wikipedia pages case study, 43

directed multigraphs, 19
directed network analysis, game developer example, 204–208
discreteness, 2
dissortative networks, 97
distance, *see also* similarity
angular distance, 172
bipartite networks, 180–181
cosine distance, 171–173, 187–192
directed graphs, 201
Euclidean distance, 171
Hamming distance, 167–169, 180, 187–192
Manhattan distance, 169–171
understanding, 163, 167–173

documentation and communication networks, 63
draw_circular(), 26
draw_networkx(), 26–28, 158
draw_random(), 26
draw_shell(), 26
draw_spectral(), 26
draw_spring(), 26
dump(), 143
Dunbar's number, 57
duplicate nodes
deleting, 45, 110
detecting, 33
merging, 33, 108

dyads
cliques, 132
defined, 6
neighborhoods as, 86

dynamics, xv, 57
dzcnapy_plotlib module, 27

E

eccentricity, 35, 91, 201
eccentricity() method, 91
ecological networks, examples, 5
economic networks, examples, 5
edge
accessing directly, 77
defining or changing attributes, 24
storing nodes with, 20

edge lists
food and nutrient example, 22
support for, 30
using, 20, 76, 209

edges, *see also* adjacency; attributes; centrality; communities; directed edges; edge lists; incident edges; labels; preferential attachment; weight
adding duplicate, 19
adding or removing with Gephi, 31, 33
adding or removing with graph-tool, 14
adding or removing with iGraph, 13
adding or removing with networkx, 19, 23, 46, 103
assortativity, 97–100
Barabási–Albert graphs, 65, 79
in classic networks, 2–4
co-occurrence, 115, 119
communication networks, 61
converting similarities to, 163
core-peripheral analysis, 130
defined, 2
density, 84
directed graphs, 18, 199
distinguishing strong and weak in social networks, 66
Erdös–Rényi graphs, 64, 79
event networks, 164
gathering for Wikipedia pages case study, 41–44
Holme–Kim graphs, 65, 79
incidence matrices, 76
induced, 137
isolates, 125
measuring network size, 83
merging duplicates, 45
negatively weighted, 116, 120, 126
networkx 2.0, 213
non-existent, 84, 134
parallel edges and multigraphs, 18–19
path length, 89
product networks, 120–123
reversing, 202
selecting, 23
self-loops, 18, 33, 45, 89
semantic networks, 116–117, 119
signed, 199
social networks, 53, 57–60, 66
storing in networkx, 20
synthetic networks, 63–66, 78
trails, 89
truncating networks, 46
undirected graphs, 18
walks, 89
Watts–Strogatz graphs, 64, 79

edges attribute, 213
edges() method, 20, 77, 84, 213
EdgeView, 213
ego networks
clustering coefficient, 87
defined, 54
Facebook example, 55–57
neighborhood, 84–86
Panama Papers case study, 109
understanding, 54–57

ego nodes, 54–57, 84–87, 89
ego_graph(), 86
egocentric networks, *see* ego networks

eigenvector centrality
  defined, 94
  directed graphs, 201
  measuring, 35, 96, 201
  Othello semantic network, 119
  product networks, 122
  social networks, 57
  spectral layout, 26
eigenvector_centrality(), 96
email as communication network, 62
empiric networks, 61
empty graphs, 18
Enron, 62
entities, *see* nodes
erdos_renyi_graph(), 78
Erdös–Rényi graphs
  about, 64
  generating synthetic networks, 78
  sliced example, 80
ERGMs (exponential random graph models), xv, 6
Euclidean distance, 171
event networks, example of similarity, 164–166
evolution, xv, 6
Exploratory Social Network Analysis with Pajek, 58
exponential random graph models (ERGMs), xv, 6
exporting, networks, 30, 32, 39

F

Facebook
  ego network, 55–57
  empiric networks, 61
  median number of friends, 55
  size of social network, 57
  as social networking website, 54
family networks, *see also* Abraham Lincoln timeline
  directed graphs, 18
  Florentine families synthetic network, 65, 79
family trees, defined, 3
Federal Energy Regulatory Commission, 62
files, *see also* CSV files
  formats for saving visualizations, 32, 39
  importing and exporting networks, 30
filtering
  components, 104
  with Gephi, 32
  with graph-tool, 14
.find(), 13
find_cliques(), 132, 134
flattening, 202
Florentine families synthetic network, 65, 79
florentine_families_graph(), 79
flows, xv, 198
food and nutrient examples
  adding and removing nodes and edges, 19–23
  adding attributes, 23–25
  bipartite networks, 175–183
  building programmatically, 19–30
  with Gephi, 32–40
  reading CSV file, 21
  saving network, 29–30
  sketching by hand, 6–8
  visualization with Gephi, 37–40
  visualization with matplotlib, 25–28
food fraud semantic network example, 116–118
food pantry examples
  binarizing attributes, 166
  as product network, 120
  distance, 167–171
  similarity, 166–171
force-directed layout, *see* Fruchterman-Reingold layout
ForceAtlas2 layout, 37
friendship paradox, 57
from_dict_of_lists(), 77
from_edgelist(), 76
from_numpy_matrix(), 72–73
from_pandas_adjacency(), 214
from_pandas_dataframe(), 73, 180, 214
from_pandas_edgelist(), 214
frozenset(), 135
Fruchterman-Reingold layout, 26–28, 37
fruchterman_reingold_layout(), 26

G

game developer language
  toposort example, 204–208
GCC (giant connected component), 125, 128–129, 155
generalized module, xv, 182
generalized similarity, 174, 181, 187–192
generalized_similarity module, 182
geodesics
  betweenness centrality, 93
  closeness centrality, 93
  defined, 90
  eccentricity, 91
Gephi
  about, xiv, 31
  arrows in, xvi
  capabilities, 31
  creating networks, 31
  cultural domain analysis, 148
  directed graphs, 199
  installing, 31
  integrating with networkx, 40
  limitations, 32
  main window and tabs, 32
  measuring connectedness, 32, 35
  network analysis with, 32, 34–37
  Panama Papers case study, 110
  passing and saving networks, 29, 31–33, 39
  saving visualizations, 32, 38–39
  using, 31–40
GetNet, 55
giant connected component (GCC), 125, 128–129, 155
The Good Wife case study, 141–151, 176
Google+, empiric network, 61
Granovetter, Mark, 66
graph exchange XML format, importing and exporting, 30
graph modeling language, importing and exporting, 30
Graph(), 18

graph-tool, 11–13, 16, 211
GraphML
  dictionary attributes, 45
  importing and exporting, 30, 40
graphs, *see also* directed graphs; induced graphs; multigraphs; networks; undirected graphs
  defined, 2
  empty, 18
  pseudographs, 18, 53
  relabeling nodes, 22
  simple network example, 18
  types, 18
graphviz
  about, 28
  cosmetics case study, 158
  food and nutrient example, 28
  with graph-tool, 14
  installing, xvi
  node placement in networkx, 13
graphviz-dev, xvi
grid_2d_graph(), 78
grids
  defined, 3
  synthetic networks, 64, 78
groupby(), 157
groups, *see* modularity

H

Hamming distance, 167–169, 180, 187–192
hamming(), 168
harmonic centrality, 93, 96
harmonic_centrality(), 96
height, specifying for trees, 78
HeightsWeights dataset, 170
hierarchical networks
  defined, 4
  directed acyclic graphs, 203
  in social network analysis, 6
HITS (Hyperlink-Induced Topic Search), 35, 95–96
hits(), 96
Holme–Kim graphs, 65, 78
homophily, 57, 98
hubs, 35, 95–96
hypergraphs, 2
Hyperlink-Induced Topic Search (HITS), 35, 95–96
hypernyms, 198
hyponyms, 198

I

Iago, 119
icons, changing size and color with Gephi, 31
iGraph, 11–12, 15, 137, 210
importing
  location for imports, 102
  networks, 30–33
  node attributes into DataFrame, 74
in_degree attribute, 213
in_degree() method, 48, 199
in_degree_centrality(), 201
incidence matrices, creating networks from, 69, 76
incidence_matrix(), 76
incident edges
  defined, 17
  directed graphs, 199
  removing nodes and, 19
  selecting, 23
incident nodes
  adding edges with networkx, 103
  defined, 17
  incidence matrices, 76
indegree
  adjacency matrices, 70
  centrality, 92, 201
  defined, 46
  directed graphs, 199
  networkx 2.0, 213
  reversing, 202
  Wikipedia pages case study, 46
indexes
  contiguous indexes in graph-tool, 14
  Series index, 146
induced edges, 137
induced graphs
  bipartite networks, 178
  condensation, 202
  creating, 137
  self-loops, 157
induced nodes, degree, 159
induced_edges(), 137
induced_graph(), 137, 157
influence, 57
information, *see* knowledge
iPython, 71
is_bipartite(), 177, 187
is_directed_acyclic_graph(), 203
is_isomorphic(), 77
isolates
  defined, 125
  deleting, 126
  locating, 125
  problems representing in pure Python, 210
  product networks, 122, 125
isolates() method, 126
isomorphism, edge lists, 77
itemgetter, 42, 157
items(), 157
itertools, 153, 157

J

Justice Resource Institute, 186

K

k-clique, *see* cliques
k-core, *see* core
k-corona, *see* corona
k-crust, *see* crust
k-partite networks, *see* bipartite networks; multi-partite networks
k-shell, *see* shell
k_clique_communities(), 135
k_core(), 131
k_corona(), 131
k_crust(), 131
k_shell(), 131
Karate Club synthetic network, 65, 79
karate_club_graph(), 79
KeyError, 24
knowledge
  information dissemination/diffusion, 57, 62
  preservation and social networks, 57, 66
  representation, 116
Koblenz Network Collection (KONECT), 61
Kovacs algorithm, 181

Krevl, Andrej, 61
Kunegis, Jérôme, 61

L

labels
- Abraham Lincoln timeline, 210
- auto-generating community labels, 157–160
- editing with Gephi, 31, 33, 37
- enabling in Gephi, 33
- food and nutrient example, 24, 28
- in graph-tool, 15
- in graphviz, 28
- issues with pure Python, 210
- matplotlib and, 28
- naming extracted blocks, 139
- in networkx, 18
- processing attributes selectively, 109
- relabeling nodes, 22, 103

layout
- cosmetics case study, 158
- Gephi, 32, 37–40
- graphviz, 28, 158
- networkx, 26–28

lemmatization, 144–145
len(), 21, 83, 85
Leskovec, Jure, 61
letter format, 40
linear networks, *see* paths
links and asymmetric relationships, 198
lists, *see also* edge lists
- cliques, 132
- connected components, 127
- empty list accumulator, 145
- importing and exporting, 30
- list of lists, converting to, 72
- list of lists, to represent matrices, 70
- measuring length, 21
- neighborhood measurement, 85
- node dictionaries, 76–77
- removing non-existent nodes or edges, 19
- speed, 25
- term lists in cultural domain analysis, 142, 144
- term vectors, 145

LiveJournal
- about, 142
- case study, 141–151, 176
- empiric network, 61

local bridges, 94
local topology, examples, 57
log(), 106
long tail in power law distribution, 107
louvain algorithm, 136–137
lower(), 145

M

machine learning and naming extracted blocks, 139
main core, 130
make_max_clique_graph(), 133
Manhattan distance, 169–171
marketing analysis, 122
Markov chains, 95
Mathematica, xiii
matplotlib
- about, 71
- food and nutrient example, 25–28
- importing, 26
- integration with networkx, 25
- Panama Papers case study, 101
- plotting centralities, 97
- version, xv

matrices
- attribute mixing, 105–107
- bi-adjacency matrix, 180, 187
- creating networks from adjacency and incidence matrices, 69–76
- dense, 76
- event networks, 166
- list of lists to represent, 70
- list of lists, converting to, 72
- matrix-multiplying, 147
- similarity matrix, 188
- sparse, 76, 147
- term matrix in cultural domain analysis, 144
- unit, 96

matrix-multiplying, 147
maximal cliques, 132–133, 138
maximum cliques, 132
mean reciprocal distance and closeness centrality, 93
measurement, 83–100, *see also* degree; distance
- centrality, 32, 35, 90, 92–97, 201
- clustering coefficients, 32, 35, 87
- degree centrality, 35, 92, 96, 201
- directed graphs, 199–208
- eccentricity, 35, 91, 201
- estimating uniformity through assortativity, 97–100
- with Gephi, 32, 34–37
- global measures, 83
- neighborhoods, 84–88
- paths, 32, 35, 88–92, 131
- similarity, 163–164, 167–173
- size, 83
- Wikipedia pages case study, 46, 83–100

Measuring Tie Strength in Implicit Social Network, 118
membership and asymmetric relationships, 198
memory
- graph-tool, 14
- incidence matrices, 76
- measuring non-existent edges, 84

merging, duplicate nodes, 33, 108
meshes, *see* grids
migration distribution example of directed graphs, 199–204
modularity
- cosmetics case study, 157
- cultural domain analysis case study, 148
- defined, 136
- measuring, 32, 35
- modularity classes, 35
- outlining modularity-based communities, 136–138

partitioning networks into communities, 35
trauma types case study, 191

modularity() method, 137

modules
installing, xvi
versions used in this book, xv

MoiKrug, 60

monads, cliques, 132

Moreno, J.L., 5

Mossack Fonseca, 101

most_common(), 158

multi-partite networks
about, 176
anti-communities, 136
examples, 176

MultiDiGraph(), 19

MultiGraph(), 18

multigraphs
adding duplicate nodes or edges, 19
adjacency matrices, 69
creating, 18
defined, 18
directed, 19
social networks, 53

MySpace, 55

N

names
cosmetics case study, 157–160
extracted blocks, 139
merging duplicate, 108
name generators, 54
term communities, 148–150

NaN (not a number), 75, 109, 165

natural language processing
cultural domain analysis case study, 142–151
distinguishing strong and weak edges in social networks, 66
lemmatizing, 144–145
stemming, 145
stop words, 42, 144–145
toposort example, 204–208
Zipf's law, 147

Natural Language Toolkit
cultural domain analysis case study, 142–151
version, xv

neighborhoods
assortativity, 98
defined, 84
directed graphs, 200
as dyadic, 86
measurement, 84–88
in networkx 2.0, 213
open, 85
path length, 89
transitive closure, 5

neighbors() method, 200, 213

Network Analysis, 64

Network Analysis of Exposure to Trauma and Adverse Events in a Clinical Sample of Children and Adolescents, 185

network dynamics, xv

network flows, xv

Network Science, 185

NetworKit, 11–13, 15, 212

networks, *see also* assortativity; bipartite networks; classic networks; clustering coefficient; complex networks; eccentricity; ego networks; graphs; measurement; neighborhoods; semantic networks; signed networks; similarity; simple networks; social networks; synthetic networks
as circles, 91
communication, 61
creating from adjacency and incidence matrices, 69–76
creating with Gephi, 31
defined, 2–4
dissortative, 97
editing with Gephi, 31
empiric, 61
event networks, 164–166
flows, 198
hierarchical, 4, 6, 203
importing and exporting, 30–33, 39
saving, 29–30
scale-free, 6
separating cores, shells, coronas, and crusts, 129–131
small-world, 6, 57, 64, 78
splitting into connected components, 126–128
structural elements, 125–139
truncating, 46

Networks, Crowds, and Markets, 185

networkx
about, xiii, 1, 11
Abraham Lincoln timeline example, 212
adding attributes, 23–25
adding or removing nodes and edges, 19, 23, 46, 103
advantages, 15
bipartite networks, 177–178, 180
building food and nutrient example programmatically, 19–30
centrality measurement, 92–97
cliques, 132–135
clustering coefficient measurement, 87
community detection, 12
component analysis, 127
converting Panama Papers CSV file with Pandas, 108–111
converting Panama Papers CSV file without Pandas, 101–107
converting adjacency matrices, 71–75
core-peripheral analysis, 131
cosmetics case study, 153–160
cultural domain analysis case study, 147
density measurement, 84
directed acyclic graphs, 203–208
directed graphs, 199–208
eccentricity measurements, 91
edge lists and node dictionaries, 76–77
estimating uniformity through assortativity, 97–100
generating synthetic networks, 78
global measures, 83
graph creation, 18
graphviz and node placement, 13
importing, 17

- importing and exporting networks, 30
- integration with Gephi, 40
- integration with matplotlib, 25
- isolates, 126
- layout options, 26–28
- modularity-based communities, 136–138
- neighborhood measurement, 85
- node and edge lists, 20
- path measurements, 88–92
- reading CSV file, 21
- relabeling nodes, 22
- resources on, 12, 15, 213
- saving and sharing networks, 29–30
- saving visualizations, 26
- similarity, 164
- simple network with, 17–30
- size measurement, 83
- slicing weighted networks, 79–81
- speed, 13, 15
- stubs for directed edges visualization, xvi
- version 1.11, xv
- version 2.0, 208, 213
- visualization size limitations, 110
- visualization with matplotlib, 25–28
- weight assumptions, 71

NetworkX Google discussion group, xvii

NetworkXError, 19

Newman's definition, 136

next(), 90, 155

nltk
- cultural domain analysis case study, 142–151
- version, xv

node
- defining or changing attributes, 24
- degree, 35
- storing nodes with, 20

node dictionaries
- CSV lookup, 103
- moving data with, 76–77

nodes, *see also* adjacency; attributes; centrality; cliques; communities; degree; incident nodes; isolates; labels; neighborhoods; preferential attachment; snowballing
- adding duplicate, 19
- adding or removing with Gephi, 31, 33
- adding or removing with graph-tool, 14
- adding or removing with iGraph, 13
- adding or removing with networkx, 19, 23, 46, 103, 110
- alter nodes, 54–57, 84–86
- assortativity, 97–100
- avoiding merging, 158
- Barabási–Albert graphs, 65, 79
- bipartite networks, 177–183
- in classic networks, 2–4
- core, 47
- core-peripheral analysis, 130
- defined, 2
- detecting duplicate nodes, 33
- directed graphs, 199
- discovering new nodes in ego networks, 55
- discreteness, 2
- dyadic, 6
- editing in Gephi, 33–37
- ego nodes, 54–57, 84–87, 89
- Erdös–Rényi graphs, 64, 79
- gathering for Wikipedia pages case study, 41–44
- Holme–Kim graphs, 65, 79
- identifying, 7
- induced, 159
- measuring network size, 83
- measuring number of, 21
- merging duplicate, 33, 108
- naming in cosmetics case study, 157–160
- networkx 2.0, 213
- path length, 89
- product networks, 120–123
- random node sampling, 59
- reflexive, 2
- removing duplicate, 110
- removing from ego networks, 55
- seed, 7, 41, 59, 109
- self-loops, 18
- semantic networks, 117, 119
- similarity-based networks, 163–174
- social networks, 53, 57–60
- source nodes, 69, 90
- splitting in bipartite networks, 177
- storing in networkx, 20
- supernodes, 138
- synthetic, 133, 137–138
- synthetic networks, 61, 63–66, 78
- target nodes, 69, 90
- transitive closure, 5
- triadic, 6
- truncating networks, 46
- Watts–Strogatz graphs, 64, 79

nodes attribute, 213

nodes() method, 20, 213

NodeView, 213

non-existent edges, 84, 134

non_edges() method, 84

number_of_edges(), 83

number_of_nodes(), 83

NumPy
- about, 71
- assortativity, 99
- converting adjacency matrices, 71
- cultural domain analysis case study, 142–151
- generating unit matrix, 96
- Hamming distance conversion, 169
- random layout and, 26
- version, xv

nutrient examples, *see* food and nutrient examples

O

Odnoklassniki, 59, 97–100

Oh No They Didn't! (blog), 142

open neighborhoods, 85

OpenMP, 13, 16

ordering
  asymmetric relationships, 198
  lookup, 43
  in networkx 2.0, 214
  selecting attributes by area of interest, 109
Othello, 118
out_degree attribute, 213
out_degree() method, 48, 199
out_degree_centrality(), 201
outdegree
  adjacency matrices, 70
  centrality, 92, 201
  defined, 46
  directed graphs, 199
  networkx 2.0, 213
  reversing, 202
Overview tab in Gephi, 32

P

p-value, 173, 189
PageRank, 35, 94–96
pagerank(), 95–96
painting project example, 122
Pajek, xiii, 30, 148
Panama Papers case study, 101–111, 176
Pandas
  about, 71, 73
  binarizing attributes, 166
  centrality measurements, 96
  converting Panama Papers CSV file, 108–111
  converting adjacency matrices, 71, 73–75
  cosine similarity, 172
  cultural domain analysis case study, 142–151
  modularity-based communities, 137
  networkx 2.0, 214
  Pearson correlation, 174, 180
  similarity, 164
  version, xv
paper and pencil sketching networks, 6–8
parallel edges, 18–19
parallelism
  with graph-tool, 13
  with NetworKit, 15
Pareto principle, 57, 129, 147
partitioning
  clique communities and, 135
  cosmetics case study, 157
  modularity-based communities, 136–138, 191
  networks into communities, 35, 88, 148–150, 157
  networks into connected components, 126–128
  nodes in bipartite networks, 177
  term communities, 148–150
  trauma types case study, 191
path length
  cliques, 131
  components, 131
  measuring, 32, 35, 88–92
path_graph(), 78
paths
  connected components, 126
  cutoff for shortest, 109
  directed graphs, 201
  geodesics, 90
  measuring, 32, 35, 88–92, 131
  synthetic networks, 64, 78
  as trails, 89
PDF files, saving as, 32, 39
Pearson correlation, 173, 180, 182, 187–192
pearsonr(), 173
pecking order, 203
periphery
  core-peripheral analysis, 129
  crust as, 130
  defined, 91, 129
  directed graphs, 201
  eccentricity, 91
  measuring, 91, 201
periphery() method, 91
pickle
  cultural domain analysis case study, 143
  importing and exporting networks, 30
plot(), 27
plug-ins and Gephi, 32
PNG files, saving as, 32, 39
power law distribution, 106, 147
powerlaw_cluster_graph(), 78
pre-painting project example, 122
predecessors, 200, 202
predecessors() method, 200
preferential attachment
  Barabási–Albert graphs, 65, 78, 106
  defined, 5
  giant connected component (GCC), 129
  network dynamics, 57
Preview tab in Gephi, 32, 38
product networks
  bipartite network example, 179
  cliques, 133
  cosmetics case study, 153–160
  defined, 120
  food pantry example, 120
  isolates, 125
  understanding, 120–123
projected_graph(), 179
projections
  bipartite networks, 178–183, 190
  event networks, 164
properties, node, *see* attributes
property maps, 15
pseudographs, 18, 53
pure bridges, 94
pygraphviz, version, xv
pyplot submodule, 26
Pythagoras' Trousers, 171
Python, version, xv
python-louvain, 136

Q

Qualtrics, 205
quotient_graph(), 214

R

R, iGraph support, 12
radius, 91, 201
radius() method, 91
random layout, 26–28
random node sampling, social networks, 59
random_graph(), 178

random_layout(), 26
Read, Ronald C., 78
read_adjlist(), 30
read_edgelist(), 30
read_gexf(), 30
read_gml(), 30
read_graphml(), 30
read_pajek(), 30
read_pickle(), 30
read_yaml(), 30
reciprocal mean distance and closeness centrality, 93
reflexive nodes, 2
regularity and simple networks, 4
relabel_nodes(), 22
relationships, *see* edges
remove_edge(), 20
remove_edges_from(), 20, 22
remove_node(), 20
remove_nodes_from(), 20
renderers, Gephi, 39–40
replace(), 108
resources for this book
    code files, xvii
    networkx, 12, 15, 213
    online communities, xvii, 12, 15
reversal, 95, 202, 213
reverse(), 95, 202, 213
rings
    as bipartite network, 175
    defined, 4
    synthetic networks, 64, 78

S

sampling
    random node sampling, 59
    snowballing, 7, 41–44, 59
saving
    networks, 29–30
    unwanted nodes in ego networks, 55
    visualizations in Gephi, 32, 38–39
    visualizations in networkx, 26
scale-free networks, 6
scaling
    Fruchterman-Reingold layout in Gephi, 37
    networkx issues, 27
    scale-free networks, 6
SciPy
    about, 71, 73
    cosine similarity, 172
    Hamming distance, 168
    Manhattan distance, 170
    Pearson correlation, 173, 180
    version, xv
search
    breadth-first search, 43
    iGraph, 13
seed nodes
    Panama Papers case study, 109
    random sampling with, 59
    snowballing with, 7, 41
.select(), 13
selection operator ([]), 23
self-loops
    adjacency matrices, 69
    as cycle, 89
    deleting edges, 45
    identifying, 22
    induced graphs, 137, 157
    merging duplicates and, 33, 45
    networkx 2.0, 214
    removing, 22, 45
    undirected graphs, 18
selfloop(), 214
selfloop_edges(), 22
semantic domain analysis, 116
semantic networks
    asymmetric, 198
    cliques, 133
    defined, 116
    food fraud example, 116–118
    isolates, 125
    Othello example, 118
    understanding, 116–120
sentiment analysis, 66
Sephora cosmetics case study, 153–160
serialization
    directed acyclic graphs, 203
    with pickle, 143
Series
    building, 75
    defined, 73
    importing node attributes, 75
    index, 146
    joining in DataFrame, 146
    modularity-based communities, 137
    term vector model, 146
Series index, 146
set_edge_attributes(), 24, 214
set_extent(), 27
set_node_attributes(), 24, 214
sets
    bipartite networks, 177, 213
    frozen sets, 135
    speed and, 25
shared memory multiprocessing, 13
shell
    defined, 130
    deleting all nodes and edges while keeping, 20
    separating, 129–130
shell layout, 26
shell_layout(), 26
shortest_path(), 90
shortest_simple_paths(), 90
signed edges, 199
signed networks
    adjacency matrices, 69
    defined, 60
    vs. directed networks, 199
    weight, 60, 71, 199
similarity, 163–174, *see also* distance
    bipartite networks, 180–183
    converting similarities to edges, 163
    cosine, 171–173, 187–192
    generalized, 174, 181, 187–192
    local topology and, 57
    matrix, 188
    measuring, 163–164, 167–173
    Pearson correlation, 173, 180, 182, 187–192
    trauma types case study, 185–192
    understanding, 163–167
similarity matrix, 188
similarity_mtx(), 188

similarity_net(), 189
simple networks, *see also* classic networks
  adjacency matrices, 69
  clustering coefficient, 88
  with networkx, 17–30
  regularity and, 4
single_source_shortest_path_length(), 109
sinks, 95
six degrees of separation, *see* small-world networks
sketching networks by hand, 6–8
SlashDot, empiric network, 61
slice_projected(), 190
slicing
  bipartite networks, 180, 183, 189
  cultural domain analysis case study, 147
  defined, 80
  with graph-tool, 14
  Hamming distance, 169
  isolates and, 126
  with networkx, 79–81
  product networks, 120
  similarity matrix, 189
  threshold, 80, 126, 147, 180, 183, 189
small-world networks, 6, 57, 64, 78
SNA, *see* social network analysis
snowballing
  defined, 7
  food and nutrient sketch example, 7
  social networks, 59
  Wikipedia pages case study, 41–44
Social and Economic Networks, 92
social capital, 57
Social Network Analysis, 54, 58
social network analysis, *see also* social networks
  clustering coefficient, 87
  core-peripheral analysis, 129
  eccentricity, 92
  history, 5
  neighbors, 84–88
social networking websites
  empiric networks, 61
  vs. social networks, 54
social networks
  acquiring, 59–60
  asymmetric, 197
  communication networks, 61
  core-peripheral analysis, 129
  defined, 5, 53, 57
  distinguishing strong and weak edges, 66
  examples, 5
  neighborhoods measurement, 84–88
  Othello semantic network, 119
  path length, 89
  prepared, 61
  properties table, 57
  signed networks, 60
  vs. social networking websites, 54
  synthetic networks, 63–66
  understanding, 53–67
sociocentric networks, *see* social networks
sociograms, 5
SOCR Data Dinov 020108 HeightsWeights dataset, 170
sorting, tuples, 42
source nodes, 69, 90
Southern women synthetic network, *see* Davis Southern women synthetic network
sparse matrices, 76, 147
spectral layout, 26–28
spectral_layout(), 26
speed
  calculating generalized similarity in bipartite networks, 182–183
  cliques, 132
  component analysis, 128
  converting adjacency matrices, 71–72
  graph-tool, 13–14, 16
  iGraph, 12, 16
  lists, 25, 132
  NetworKit, 13, 16
  networkx, 13, 15
  problems with pure Python, 210
  snowballing, 43
spring layout, 26–28
spring_layout(), 26
Stack Overflow forums, xvii, 12, 15
Stanford Large Network Dataset Collection, 61
star_graph(), 78
stars
  clustering coefficient, 87
  defined, 4
  preventing with stop words, 42
  synthetic networks, 64, 78
statistics
  calculation tools with graph-tool, 14
  with Gephi, 32, 35
stemming, 145
stop words
  cultural domain analysis case study, 144–145
  preventing stars when snowballing, 42
strongly_connected_component_subgraphs(), 128
strongly_connected_components(), 127
subgraph(), 46, 128, 213
subgraphs
  connected components, 128
  ego networks, 56
  in networkx 2.0, 213
  truncating network, 46
subgraphs, complete, *see* cliques
subordination, 197, 203
subsets
  clustering, 88
  defined, 6
substitutes, product networks, 120
successors, 200, 202
successors() method, 200
supernodes, synthetic, 138
SurveyMonkey, 205
SVG files, saving as, 32, 39
symmetry, 18
SymPy, 71

synthetic networks
  complex, 78
  generating, 61, 78
  regular, 78
  understanding, 63–66
synthetic nodes
  blockmodeling with synthetic supernodes, 138
  modularity-based communities, 137
  replacing maximal cliques with, 133

T

target nodes, 69, 90
technological networks, examples, 5
term communities, portioning and naming, 148–150
term matrices, cultural domain analysis, 144
term vector model (TVM), 146
term vectors, 145
terms
  cultural domain analysis, 142, 144
  extracting and naming term communities, 148–150
  term lists in cultural domain analysis, 142
  term vector model (TVM), 146
  term vectors, 145
ties, *see* edges
timelines, defined, 3, *see also* Abraham Lincoln timeline
to_dict_of_lists(), 77
to_directed(), 213
to_edgelist(), 76
to_numpy_matrix(), 72–73
to_pandas_adjacency(), 214
to_pandas_dataframe(), 73, 214
to_pandas_edgelist(), 214
to_undirected(), 202, 213
todense(), 76
tolist(), 72
top nodes, bipartite networks, 178–183
topological_sort(), 203, 206
topology
  directed acyclic graphs, 203–208
  examples, 57
toposort module, xv, 204–208
toposort() method, 207
trails, 88, 201
transitive closure, 5, 88, 203
transitive_closure(), 203
transitivity(), 88
trauma types case study, 185–192
trees
  branching factor, 78
  defined, 3
  stars, 4
  synthetic networks, 64, 78
tri(), 96
triadic census, xv
triadic closure, 57
triads
  cliques, 132
  clustering coefficient measurement, 87
  defined, 6
tripartite networks, examples, 176
truncating networks, 46
tuples, sorting, 42
TVM (term vector model), 146
Twitter, empiric network, 61
two-mode networks, *see* bipartite networks

U

UCINET, 148
unconnected graphs and snowballing, 7
undirected graphs
  adjacency matrices, 69
  converting directed graphs to, 18, 128, 202
  creating, 18
  defined, 18
  density, 84
  networkx 2.0, 213
  social networks, 53
unit matrices, generating, 96
United States Census Bureau State-to-State Migration Flows dataset, 199
United States Department of Agriculture (USDA), 121, 166
urllib.request module, 154
USDA (United States Department of Agriculture), 121, 166

V

versions
  charting with networkx, 13
  modules used in this book, xv
  networkx, xv, 208, 213
  Python, xv
views
  with graph-tool, 14
  networkx 2.0, 213
visualizations
  classic networks, 2–4
  directed graphs, 199
  with Gephi, 31–40, 199
  graphviz, 28
  layout options, 26–29, 32, 37–40
  layout phase, 26
  with matplotlib, 25–28
  rendering phase, 26
  saving in Gephi, 32, 38–39
  saving in networkx, 26
  scaling, 27
  size limits of networkx, 110
  sketching by hand, 6–8
  tools, 11–16

W

walks, 88, 201
Watts–Strogatz graphs, 64, 78
weakly_connected_component_subgraphs(), 128
weakly_connected_components(), 127
weight
  adding weighted edges, 24
  adjacency matrices, 69
  assumptions in networkx, 71
  bipartite networks, 179–180, 189
  bridges, 66
  cliques, 66
  converting to dictionary, 147
  defined, 24
  directed networks, 199

distinguishing strong and weak edges in social networks, 66
Hamming distance, 169
incidence matrices, 76
induced graphs, 137
negative, 116, 120, 126
path length measurement, 89
product networks, 120
signed networks, 60, 71, 199
slicing weighted networks, 79–81
social networks, 53

weighted_projected_graph(), 179

whitespace and unifying duplicate names, 108

wikipedia module
importing, 42
version, xv

Wikipedia pages case study
analysis, 47–48
centrality measurement, 93–97
clustering coefficient, 87
constructing network, 41–48
density, 84
measurement, 46, 83–100
neighborhoods, 85
path measurements, 88–92

Wilson, Robin J., 78

wind rose example of cosine distance, 171–174

WordNet, 144

wordpunct_tokenize(), 145

write_adjlist(), 30

write_edgelist(), 30

write_gexf(), 30

write_gml(), 30

write_graphml(), 30

write_pajek(), 30

write_pickle(), 30

write_yaml(), 30

Y

YAML, importing and exporting, 30

Z

Zachary's Karate Club synthetic network, 65, 79

Zipf's law, 146–147

Thank you!

How did you enjoy this book? Please let us know. Take a moment and email us at support@pragprog.com with your feedback. Tell us your story and you could win free ebooks. Please use the subject line "Book Feedback."

Ready for your next great Pragmatic Bookshelf book? Come on over to https://pragprog.com and use the coupon code BUYANOTHER2017 to save 30% on your next ebook.

Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks near water. If rash persists, see a doctor. Doesn't apply to *The Pragmatic Programmer* ebook because it's older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfaction. Increase dosage regularly.

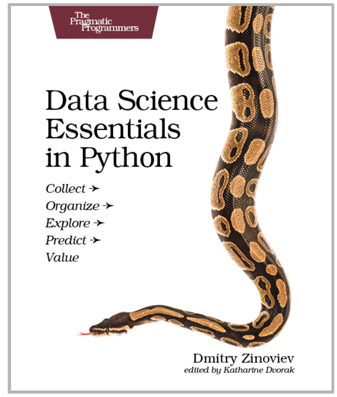
And thank you for your continued support,

Andy Hunt, Publisher

More on Python

For data science and basic science, for you and anyone else on your team.

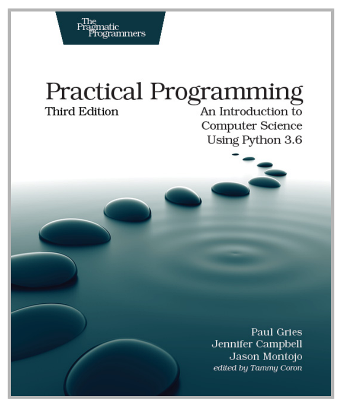
Data Science Essentials in Python



Go from messy, unstructured artifacts stored in SQL and NoSQL databases to a neat, well-organized dataset with this quick reference for the busy data scientist. Understand text mining, machine learning, and network analysis; process numeric data with the NumPy and Pandas modules; describe and analyze data using statistical and network-theoretical methods; and see actual examples of data analysis at work. This one-stop solution covers the essential data science you need in Python.

Dmitry Zinoviev
(224 pages) ISBN: 9781680501841. $29
https://pragprog.com/book/dzpyds

Practical Programming, Third Edition



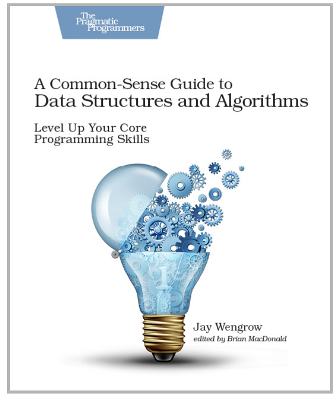
Classroom-tested by tens of thousands of students, this new edition of the best-selling intro to programming book is for anyone who wants to understand computer science. Learn about design, algorithms, testing, and debugging. Discover the fundamentals of programming with Python 3.6—a language that's used in millions of devices. Write programs to solve real-world problems, and come away with everything you need to produce quality code. This edition has been updated to use the new language features in Python 3.6.

Paul Gries, Jennifer Campbell, Jason Montojo
(410 pages) ISBN: 9781680502688. $49.95
https://pragprog.com/book/gwpy3

Level Up

From data structures to architecture and design, we have what you need.

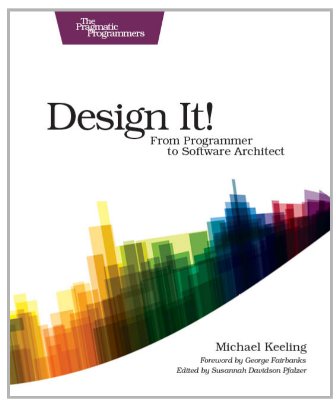
A Common-Sense Guide to Data Structures and Algorithms



If you last saw algorithms in a university course or at a job interview, you're missing out on what they can do for your code. Learn different sorting and searching techniques, and when to use each. Find out how to use recursion effectively. Discover structures for specialized applications, such as trees and graphs. Use Big O notation to decide which algorithms are best for your production environment. Beginners will learn how to use these techniques from the start, and experienced developers will rediscover approaches they may have forgotten.

Jay Wengrow
(218 pages) ISBN: 9781680502442. $45.95
https://pragprog.com/book/jwdsal

Design It!



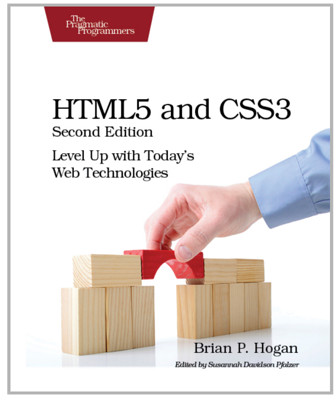
Don't engineer by coincidence—design it like you mean it! Grounded by fundamentals and filled with practical design methods, this is the perfect introduction to software architecture for programmers who are ready to grow their design skills. Ask the right stakeholders the right questions, explore design options, share your design decisions, and facilitate collaborative workshops that are fast, effective, and fun. Become a better programmer, leader, and designer. Use your new skills to lead your team in implementing software with the right capabilities—and develop awesome software!

Michael Keeling
(358 pages) ISBN: 9781680502091. $41.95
https://pragprog.com/book/mkdsa

The Modern Web

Get up to speed on the latest HTML, CSS, and JavaScript techniques, and secure your Node applications.

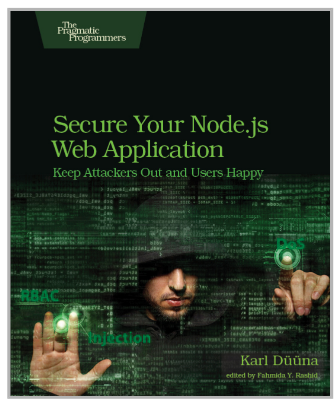
HTML5 and CSS3 (2nd edition)



HTML5 and CSS3 are more than just buzzwords – they're the foundation for today's web applications. This book gets you up to speed on the HTML5 elements and CSS3 features you can use right now in your current projects, with backwards compatible solutions that ensure that you don't leave users of older browsers behind. This new edition covers even more new features, including CSS animations, IndexedDB, and client-side validations.

Brian P. Hogan
(314 pages) ISBN: 9781937785598. $38
https://pragprog.com/book/bhh52e

Secure Your Node.js Web Application



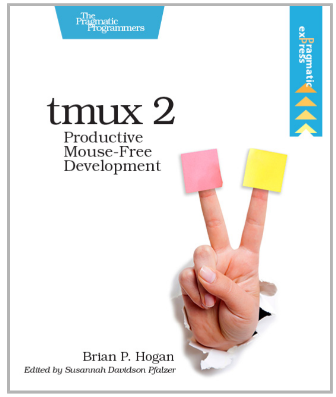
Cyber-criminals have your web applications in their crosshairs. They search for and exploit common security mistakes in your web application to steal user data. Learn how you can secure your Node.js applications, database and web server to avoid these security holes. Discover the primary attack vectors against web applications, and implement security best practices and effective countermeasures. Coding securely will make you a stronger web developer and analyst, and you'll protect your users.

Karl Düüna
(230 pages) ISBN: 9781680500851. $36
https://pragprog.com/book/kdnodesec

Long Live the Command Line!

Use tmux and Vim for incredible mouse-free productivity.

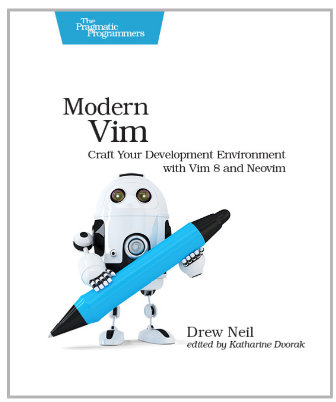
tmux 2



Your mouse is slowing you down. The time you spend context switching between your editor and your consoles eats away at your productivity. Take control of your environment with tmux, a terminal multiplexer that you can tailor to your workflow. With this updated second edition for tmux 2.3, you'll customize, script, and leverage tmux's unique abilities to craft a productive terminal environment that lets you keep your fingers on your keyboard's home row.

Brian P. Hogan
(102 pages) ISBN: 9781680502213. $21.95
https://pragprog.com/book/bhtmux2

Modern Vim



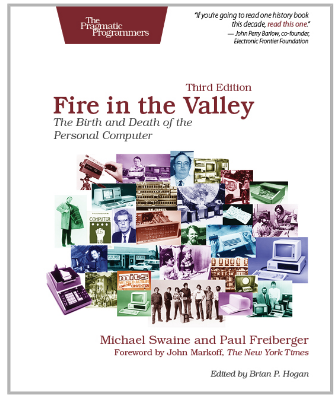
Turn Vim into a full-blown development environment using Vim 8's new features and this sequel to the beloved bestseller *Practical Vim*. Integrate your editor with tools for building, testing, linting, indexing, and searching your codebase. Discover the future of Vim with Neovim: a fork of Vim that includes a built-in terminal emulator that will transform your workflow. Whether you choose to switch to Neovim or stick with Vim 8, you'll be a better developer.

Drew Neil
(190 pages) ISBN: 9781680502626. $39.95
https://pragprog.com/book/modvim

Past and Present

To see where we're going, remember how we got here, and learn how to take a healthier approach to programming.

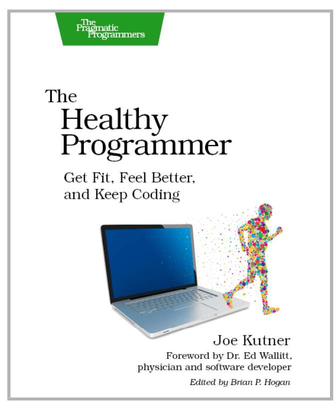
Fire in the Valley



In the 1970s, while their contemporaries were protesting the computer as a tool of dehumanization and oppression, a motley collection of college dropouts, hippies, and electronics fanatics were engaged in something much more subversive. Obsessed with the idea of getting computer power into their own hands, they launched from their garages a hobbyist movement that grew into an industry, and ultimately a social and technological revolution. What they did was invent the personal computer: not just a new device, but a watershed in the relationship between man and machine. This is their story.

Michael Swaine and Paul Freiberger
(422 pages) ISBN: 9781937785765. $34
https://pragprog.com/book/fsfire

The Healthy Programmer



To keep doing what you love, you need to maintain your own systems, not just the ones you write code for. Regular exercise and proper nutrition help you learn, remember, concentrate, and be creative—skills critical to doing your job well. Learn how to change your work habits, master exercises that make working at a computer more comfortable, and develop a plan to keep fit, healthy, and sharp for years to come.

This book is intended only as an informative guide for those wishing to know more about health issues. In no way is this book intended to replace, countermand, or conflict with the advice given to you by your own healthcare provider including Physician, Nurse Practitioner, Physician Assistant, Registered Dietician, and other licensed professionals.

Joe Kutner
(254 pages) ISBN: 9781937785314. $36
https://pragprog.com/book/jkthp

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page

https://pragprog.com/book/dzcnapy

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

https://pragprog.com/updates

Be notified when updates and new books become available.

Join the Community

https://pragprog.com/community

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

https://pragprog.com/news

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/dzcnapy*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764